

# Microarquitectura Orga 2

The FurfiOS Corporation

Febrero 2021

# Índice general

<b>1. Sistema de Memoria</b>	<b>3</b>
1.1. Introducción	3
1.2. Tecnologías	4
1.2.1. Tipos de Memorias	4
1.2.2. Memorias y velocidad del procesador	6
1.3. Cache	7
1.3.1. Organización de una cache	10
1.4. Políticas de reemplazo de contenido	11
1.4.1. Políticas de reemplazo de contenido	11
1.5. Coherencia	12
1.5.1. Políticas de Escritura	12
1.5.2. Snoopy protocols	14
1.5.3. Optimizaciones	15
1.5.4. Protocolo M.E.S.I.	15
1.6. Arquitecturas de Cache	18
<b>2. Intruction Level Parallelism (ILP)</b>	<b>21</b>
2.1. Pipeline	21
2.1.1. Obstáculos y Dependencias	23
2.2. Unidades de Predicción de Saltos	28
2.2.1. Predicción de saltos estática	29
2.2.2. Predicción de saltos dinámica	31
2.2.3. SPEC89	34
2.3. Arquitectura Superescalar	35
2.4. Ejecución fuera de orden	37
2.4.1. Scheduling dinámico	38
2.4.2. Manejando Excepciones	40
2.4.3. Algoritmo de Tomasulo	41
2.4.4. Ejecución especulativa - Reorder Buffer	45
2.5. Casos Prácticos	48
2.5.1. Three Cores Engine	48
2.5.2. Microarquitectura Netburst: Pentium 4	51
2.5.3. IA-64	52
2.6. Procesadores Multithreads: Hyper-Threading	54
2.7. Procesadores Multicore	56
<b>3. Preguntas de Final</b>	<b>59</b>

Este apunte fue hecho en base a las clases teóricas del profesor Alejandro Furfaro del Primer Cuatrimestre 2020, los apuntes de Gianfranco Zamboni, complementado con bibliografía:

- Tanenbaum[2]:
  - Capítulo 4: secciones 4.5, 4.6 (Microarquitectura: Predicción de Saltos, ejecución fuera de orden, ejecución especulativa)

- 
- Capítulo 5: sección 5.8 (IA-64: Itanium 2)
  - Capítulo 8: sección 8.1 (Multi-threading)
  - Stallings (10<sup>th</sup>ed) [5]:
    - Capítulo 14: sección 14.4 (ILP)
    - Capítulo 16: secciones 16.1, 16.2 (sistemas superescalares)
    - Capítulo 17: secciones 17.1, 17.2, 17.3 (Coherencia Cache y MESI)
    - Apéndice G de la 11<sup>th</sup>ed: secciones G.2, G.3, G.4 (Reorder Buffer, Tomasulo, Scoreboarding)
  - El Henessy Patterson [4]:
    - Capítulo 3: sección 3.1, 3.4, 3.5, 3.6 (Dependencias, scheduling dinámico, Tomasulo, Ejecución Especulativa (ROB))
    - Apéndice C: sección C.2 (Dependencias y Predicción de saltos)
  - El otro Henessy Patterson (el de Orga 1) [3]:
    - Capítulo 4: sección 4.8 (Dependencias de control y Predicción de Saltos)
  - El otro Tanenbaum [1]:
    - Capítulo 5: sección 5.1.5 (Excepciones precisas vs Imprecisas).

Acá tienen el *link* para editar el overleaf <https://www.overleaf.com/7111913282ydczdnmsphc>.

# Capítulo 1

## Sistema de Memoria

### 1.1. Introducción

En la figura 1.1 podemos ver un esquema en el cual a medida que nos acercamos a la cima, tenemos mayor velocidad (en términos de tiempo de acceso) y menor capacidad de almacenamiento (y mayor costo por bit). Luego, nos gustaría que los datos con los que vaya a trabajar el procesador estén lo más cerca posible, en el mejor de los casos, en los registros.

Para determinar qué datos nos conviene guardarnos en las memorias más rápidas, existe el **principio de vecindad** (o de localidad). Este es un principio que proviene de las investigaciones sobre el manejo de la memoria virtual, estudiado entre los años 50-60.

El comportamiento del software suele tener patrones que, al momento de diseñar el hardware, debemos tener en cuenta para que se ejecuten de forma eficiente. En general, vamos a descubrir que muchas de las cosas que el hardware diseña no son inventos, son simplemente observaciones de cómo se comporta el software, para luego derivar soluciones para agilizar ese comportamiento.

El mecanismo de cacheo en la memoria principal es parte de la arquitectura de una computadora, implementada en hardware y, típicamente, invisible al sistema operativo. Existen otras dos instancias en las cuales se utiliza un mecanismo de memorias en dos niveles: la memoria virtual y el cache del disco. Luego, los problemas de coherencia, políticas de reemplazo, etc que vamos a estudiar para la cache de la memoria principal van a ser de interés para el manejo de memoria virtual y cache del disco.

Un controlador de cache (o de memoria virtual) funciona mediante estos dos principios

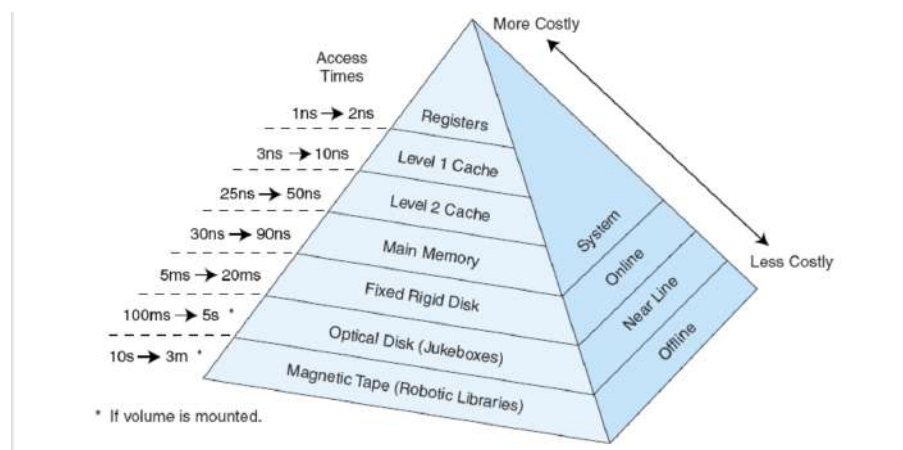


Figura 1.1: Jerarquía de Memorias

- 
- **Principio de vecindad temporal:** Una dirección de memoria que está siendo accedida actualmente tiene muy alta probabilidad de seguir siendo accedida en el futuro inmediato. Un ejemplo claro de este principio son las variables locales.
  - **Principio de vecindad espacial:** Si se está accediendo a una dirección determinada de memoria actualmente, la probabilidad de que esta dirección y sus direcciones vecinas sean accedidas en el futuro inmediato es muy alta. Por ejemplo, cuando recorremos un vector.

Estos principios establecen que los accesos a memoria tienden a agruparse. Durante un largo período de tiempo, los grupos en uso cambian, pero en un corto período de tiempo, el procesador trabaja principalmente con grupos fijos de referencias de memoria. Intuitivamente, el principio de localidad tiene sentido. Considere la siguiente línea de razonamiento:

1. A excepción de las instrucciones de *branch* y *call*, la ejecución del programa es secuencial. Por lo tanto, en la mayoría de los casos, la siguiente instrucción a buscar sigue inmediatamente a la última instrucción obtenida.
2. Es raro tener una secuencia larga e ininterrumpida de llamadas a procedimientos seguida de la correspondiente secuencia de retornos. Más bien, un programa permanece confinado a una ventana bastante estrecha en profundidad de llamadas a procedimientos. Por tanto, durante un breve período de tiempo, las referencias a instrucciones tienden a estar localizadas en unos pocos procedimientos.
3. La mayoría de las estructuras iterativas consisten en un número relativamente pequeño de instrucciones repetidas muchas veces. Por lo tanto, mientras dura la iteración, el cálculo se limita a una pequeña parte contigua de un programa.
4. En muchos programas, gran parte del cálculo implica el procesamiento de estructuras de datos, como matrices o secuencias de datos. En muchos casos, las sucesivas referencias a estas estructuras de datos serán elementos de datos ubicados cerca.

Tradicionalmente, la localidad temporal se explota manteniendo las instrucciones y los valores de datos usados recientemente en la memoria cache. La localidad espacial generalmente se explota mediante el uso de líneas de cache más grandes e incorporando mecanismos de *pre-fetching* en la lógica de controlador de la cache. Recientemente, ha habido una investigación considerable sobre el perfeccionamiento de estas técnicas para lograr un mayor rendimiento, pero las estrategias básicas siguen siendo las mismas.

## 1.2. Tecnologías

### 1.2.1. Tipos de Memorias

En principio, tenemos memorias **no volátiles**, antiguamente conocidas como memorias ROM, y se caracterizan por ser capaces de retener la información almacenada cuando se les desconecta la alimentación. Este tipo de memorias han evolucionado tecnológicamente, pasando por múltiples modelos que permitieron su modificación offline, y actualmente son modificables en tiempo real.

Se partió desde las viejas memorias denominadas ROM (Read Only Memory), que en sus primeras implementaciones debían ser grabadas durante la fabricación del chip y no eran modificables. Luego, pasaron por las memorias PROM, que se podían programar una sola vez, para luego pasar a las EPROM que podían ser borradas con luz ultravioleta. Esto siguió hasta llegar a las actuales memorias flash, que pueden ser grabadas por algoritmos de escritura *on the fly* por los usuarios, y cuyo ejemplo más habitual son las unidades de estado sólido (SSD) de los portátiles modernos.

Por otro lado, tenemos las memorias **volátiles**, antiguamente conocidas como memorias RAM (Random Access Memory), que se caracterizan por no retener la información una vez interrumpida la alimentación eléctrica. Sin embargo, estas memorias nos permiten escribir a gran velocidad en comparación a las memorias No Volátiles. Se clasifican de acuerdo con la tecnología y su diseño interno en dinámicas DRAM (memoria principal) y estáticas SRAM (cache y registros).

La memoria no volátil se usa fundamentalmente para almacenar, por ejemplo, el programa de arranque de cualquier sistema. Se conectan en un espacio de direcciones determinado por el procesador, en la que este irá a buscar la primer instrucción al encender el equipo. El resto es RAM, y allí el sistema copia incluso buena parte del código de arranque (para que se ejecute más rápido). Vamos a concentrarnos en las memorias RAM, ya que es la que utilizamos para guardar a los programas, e incluso el propio sistema operativo (una vez lo descargamos del disco).

Para empezar, definamos qué son las memorias **DRAM**. Este tipo de memorias almacenan la información en forma indirecta, a partir del estado de carga en un capacitor, y la sostiene durante un breve lapso de tiempo con la ayuda de un único transistor. Por lo tanto, la celda resulta minimalista (1 bit almacenado por transistor), dándonos la mayor densidad de hardware posible para el almacenamiento de un bit (utilizando transistores). Además, como el transistor está generalmente en estado de corte, estas celdas consumen una cantidad mínima de energía. Es decir, el transistor funciona como llave del circuito, y como normalmente está abierta, consume muy poco.

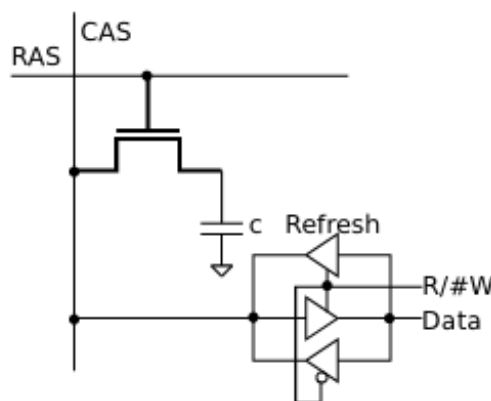


Figura 1.2: Celda DRAM

Sin embargo, la desventaja de estas celdas es que necesitan de un circuito de regeneración de carga. Cuando leemos el bit almacenado, estamos cerrando la llave, provocando que el capacitor circule una corriente, en caso de que esté cargado. Si no circuló corriente, entonces teníamos almacenado un 0, y si circuló corriente, tuvo que ceder toda la carga almacenada en el capacitor para mostrar que tenía un 1. Pero entonces, el 1 no lo leímos, nos lo llevamos en forma de corriente. Por lo tanto, necesitamos de un circuito auxiliar (6 transistores) que nos regenere la carga cada vez que leemos.

Sin embargo, estos circuitos de regeneración de carga pueden trabajar con un montón de estas celdas. De hecho, la distribución de la RAM, dentro del chip, es matricial de  $n \times n$  (supongamos  $1024 \times 1024$ ), donde por cada columna tenemos un único circuito de regeneración, y por lo tanto la cantidad de transistores que necesitamos para estos circuitos de regeneración resulta marginal respecto a la cantidad total de transistores.

Sin embargo, si bien esto no nos afecta en cuanto a la densidad de transistores por bit almacenado, sí nos afecta en cuanto a la velocidad de lectura. En general, vamos a ver que este circuito de regeneración tiene una configuración que se conoce como *push-pull*, es decir que oscila<sup>1</sup> hasta que logra restituir el valor del capacitor. Esto lleva un tiempo apreciable, y por este motivo las memorias DRAM son lentas en su tiempo de acceso. Además, debido a que la llave es una llave de silicio, tenemos una corriente de fuga, por lo que el capacitor se va descargando por sí solo, con lo cual tenemos que refrescar su valor periódicamente. Esto se hace al forzar una lectura de una fila completa, cuyo resultado descartamos.

Por su lado, las memorias **SRAM** tienen 6 transistores por cada bit almacenado. Funcionan como un bi-estable (no es otra cosa que un *latch*). Tenemos una salida  $Q$  (en la figura D) y una salida  $\bar{Q}$  (en la

<sup>1</sup>oscilar, en términos de software, sería como hacer un loop eléctrico

figura #D), pero el problema que tenemos es que cada bit de memoria requiere de 6 transistores, y por lo tanto la densidad se resigna. Además, los 6 transistores están consumiendo bastante: 3 están abiertas y 3 están cerradas. Las 3 que están cerradas hacen que circule mucha corriente, y por lo tanto hay 3 transistores que constantemente están disipando energía.

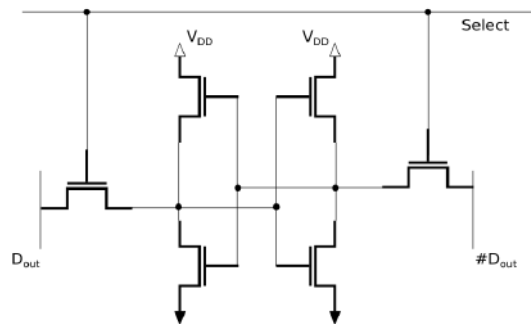
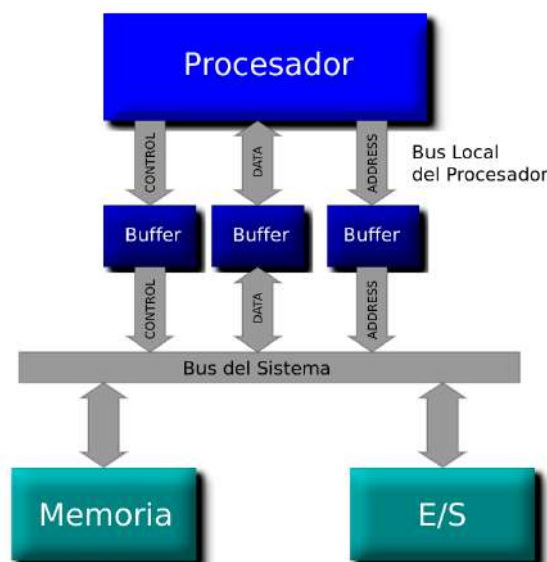


Figura 1.3: Celda de memoria SRAM

La ventaja que tienen las celdas SRAM es que, como no necesitan del mecanismo de regeneración, podemos hacer lecturas casi instantáneas.

### 1.2.2. Memorias y velocidad del procesador

Ahora veamos qué pasó históricamente, en términos relativos, entre la velocidad de los procesadores y la velocidad de las memorias. A finales de los 80, IBM presenta una PC con una arquitectura de tipo Von Neumann, con un procesador, buses, memoria, y entrada / salida.



Los buffers que vemos en la fig. 1.2.2 no son otra cosa que acondicionadores eléctricos. Un buffer permite que se le puedan conectar un bus con muchos circuitos del otro lado sin quemarlos. Cuando muchos circuitos tiran desde una salida, pueden llevar a esa salida a una sobre-corriente o puede pasar que esa salida tire un estado lógico falso (por estas eléctricamente demasiado solicitada). Lo que hace el buffer es ofrecerle a esta salida una entrada única, amplificando la corriente para que todas las demás entradas puedan tener el estado correcto. No tienen ninguna función lógica.

Ya a partir del 8088 (1979), teníamos una línea "READY", que se utilizaba para manejar memorias

---

lentas. Es decir, el procesador generaba ciclos de *wait states* para esperar a que la memoria esté lista ("READY") para el acceso. A medida que fueron apareciendo nuevos modelos, este problema en la distancia de velocidad entre el procesador y la memoria se fue agudizando (para el 80386, la velocidad del procesador ya había quintuplicado la velocidad de la memoria).

El problema, entonces, consistía en decidir qué tipo de memoria RAM usar como memoria del sistema.

- **DRAM:**

- Consumo mínimo.
- Alta capacidad de almacenamiento.
- Menor costo por bit.
- Mayor tiempo de acceso.

- **SRAM:**

- Alto consumo.
- Baja capacidad de almacenamiento.
- Costo por bit alto.
- Tiempo de acceso bajo.

Por lo tanto, si construimos un banco de memoria utilizando solo SRAM, el costo y el consumo de la computadora se va a los caños. Si construimos el banco de memoria utilizando DRAM, no aprovechamos la velocidad del procesador. La resolución de este problema viene dado por el uso de las **memorias cache**.

### 1.3. Cache

La memoria cache son un banco de SRAM de muy alta velocidad (el más rápido posible), que contiene una copia de los datos e instrucciones que están en la memoria principal. El arte consiste en que esta copia esté disponible justo cuando el procesador la necesita, permitiéndole acceder a esos datos sin recurrir a ciclos de espera. Notemos que este banco de SRAM debe ser lo suficientemente pequeño para no comprometer el costo del sistema, pero a la vez debe ser suficiente para contener siempre lo que el procesador necesita. Por ejemplo, en la primer 8038 la máquina venía con 4MB de RAM y con 4KB de memoria cache.

Luego, este pequeño banco de memoria SRAM es combinado con una gran cantidad de memoria DRAM, lo cual nos va a permitir almacenar el resto de los datos, resolviendo el problema mediante una solución de compromiso típica.

Para medir la eficiencia de la cache, tenemos las siguientes métricas:

- **Hit:** Cuando se accede a un ítem y este se encuentra en la cache, entonces tenemos un **hit**.
- **Miss:** Cuando se accede a un ítem y este no se encuentra en la cache, entonces tenemos un **miss**.
- **Hit Rate:** Luego, podemos definir al **hit rate** como

$$hit\ rate = \frac{\#hits}{\#accesos\ totales}$$

Luego, nos gustaría tener un **hit rate** lo más alto posible ( $\approx 0,9, 0,85$ ).

Para implementar un esquema de memoria con cache, se necesita de un hardware adicional: el **controlador de cache**. Este se encarga de intermediar entre el procesador y las dos memorias, asegurando que este pequeño banco de memoria cache contenga los datos e instrucciones más importantes para cuando el procesador lo necesite. Para decidir qué datos son importantes en cada momento, el controlador requiere de un criterio de decisión basado en los principios de vecindad.



Podemos ver en la figura 1.4 que el controlador de cache está conectado al bus de control y al bus de direcciones del procesador, mientras que el bus de datos pasa directamente a la memoria cache.

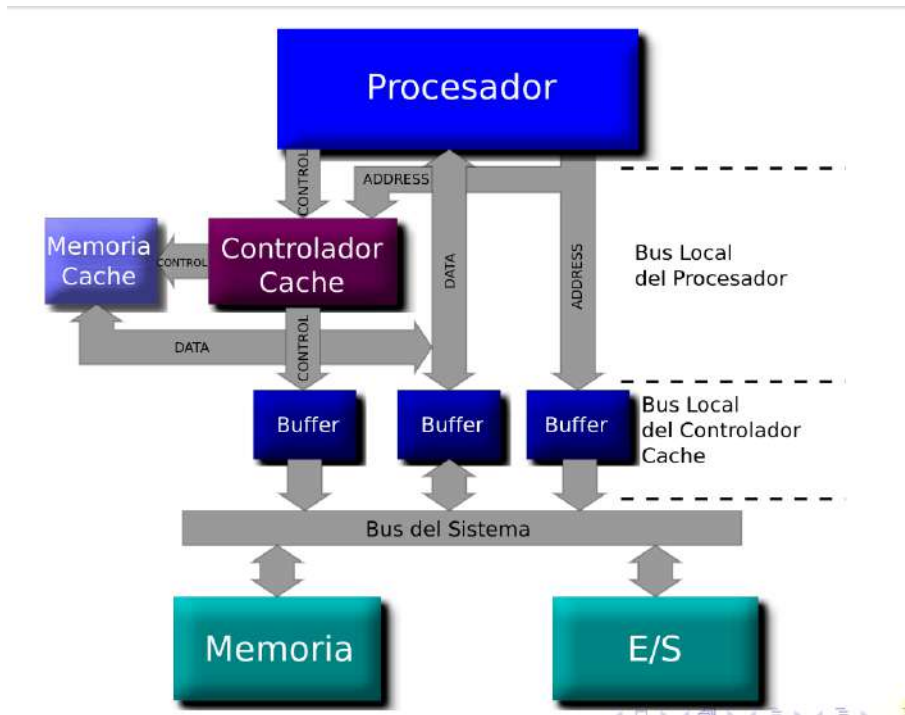


Figura 1.4: Subsistema Cache de hardware

Notemos que el bus de direcciones está conectado tanto al controlador de cache como al sistema, por lo que el controlador puede estar mirando qué direcciones está pidiendo el procesador. En cambio, el bus de control es interceptado por el controlador de cache, con lo cual el controlador de cache termina controlando al bus del sistema y no lo el procesador. El procesador dice lo que necesita y el controlador de cache se lo resuelve.

En la figura 1.5 podemos observar un diagrama de flujo para entender, de forma aproximada, cómo funciona esta cuestión.

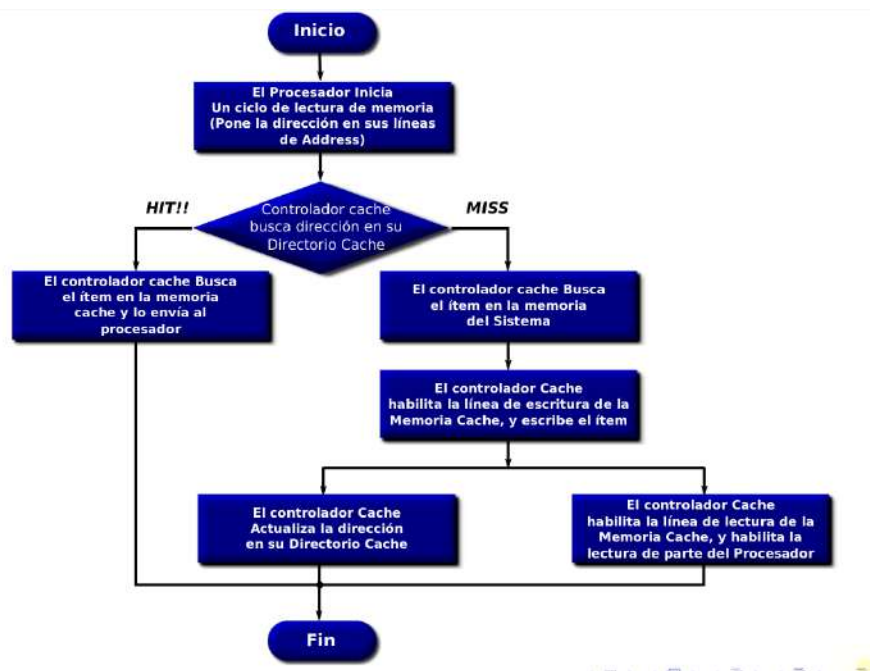


Figura 1.5: Operación de lectura.

El procesador inicia un ciclo de lectura, pone la dirección que quiere leer en las líneas de *address*, y activa la línea de control "Memory Read". El controlador de cache toma la línea "Memory Read", y también recibe por el bus de direcciones la dirección que se quiere leer. Luego, el controlador de cache revisa si esa dirección la tiene guardada, a partir de revisar si en el directorio de cache (pequeño banco de memoria SRAM) tiene esta dirección como *cacheada*. Entonces, si la encuentra, tenemos un *hit*; caso contrario tenemos un *miss*.

Si la tiene en la cache, el controlador tiene que activar las líneas de control hacia la memoria cache y no hacia el sistema. Es decir, le manda el "Memory Read" a la memoria cache y le activa el "Output Enable" para que la memoria cache envíe el resultado por el bus de datos, el cual va a ser leído por el procesador sin que pueda saber quién le envió el dato.

Por otro lado, si no la tiene, el controlador de cache se encarga de habilitar el "Memory Read" por el bus de control hacia el sistema, pidiéndole al sistema el dato. Una vez que la memoria DRAM pueda mandar la variable por el bus de datos, el controlador de cache habilita la línea "Memory Write" de la memoria cache y, como el bus de datos está conectado directamente a la memoria cache, hace que el dato nuevo se guarde en la memoria cache (vecindad temporal). Por lo tanto, la próxima vez que el procesador necesite este dato, este ya se encontrará en la memoria cache.

Luego, el controlador de cache actualiza la dirección en su directorio cache y habilita la lectura de parte del procesador para que, una vez tenga el dato en la cache, lo mande desde la cache al procesador.

Un controlador cache suele estar formado por:

- Una **interfaz con el bus local**, que actúa como árbitro del bus local.
- Una **interfaz con el procesador**, que actúa como control y decodificador del procesador.
- Un **directorio de cache**, que toma como entradas las direcciones del bus de direcciones del procesador y el bus de *snoop* (lo veremos más adelante).
- El **control de la cache**, que manda las señales de control hacia la memoria cache.

### 1.3.1. Organización de una cache

Vamos a explicar la visión que tiene el controlador de cache sobre la memoria. El controlador de cache **no** ve una sucesión de bytes, sino que ve una sucesión de líneas. Es decir, la mínima unidad de memoria para un controlador cache es una **línea**, que corresponde a algún múltiplo del tamaño de palabra del procesador (supongamos 32 bytes de tamaño de línea). En general, el tamaño de línea coincide con el ancho del bus entre la CPU y la cache, pero el bus entre la cache y la memoria principal tiene un tamaño de 64 bits, por lo que el acceso a la DRAM sigue siendo un cuello de botella.

Al trabajar con líneas y no palabras, estamos cumpliendo con el principio de vecindad espacial, el cual nos dice que si se direcciona un ítem en memoria, generalmente, se requerirá de los ítems cercanos a este. A cada línea del cache se le asigna una etiqueta (*tag*) que se obtiene a partir de la dirección de memoria **física** de la línea que contiene al dato. Además, en el directorio de cache tenemos unos bits de control: un bit de validez individual de la línea y un bit de validez general del tag (conjunto). Notemos que, dentro del cache, el orden de las líneas no tiene ninguna relación con respecto a cómo están realmente en memoria, porque las estamos identificando mediante el tag.

En general, vamos a tener un esquema parecido al que vemos en la figura 1.6. Podemos observar que el esquema se divide en tres bloques diferenciados: **directorio de cache** (amarillo), **memoria cache** (turquesa), **memoria principal** (el resto).

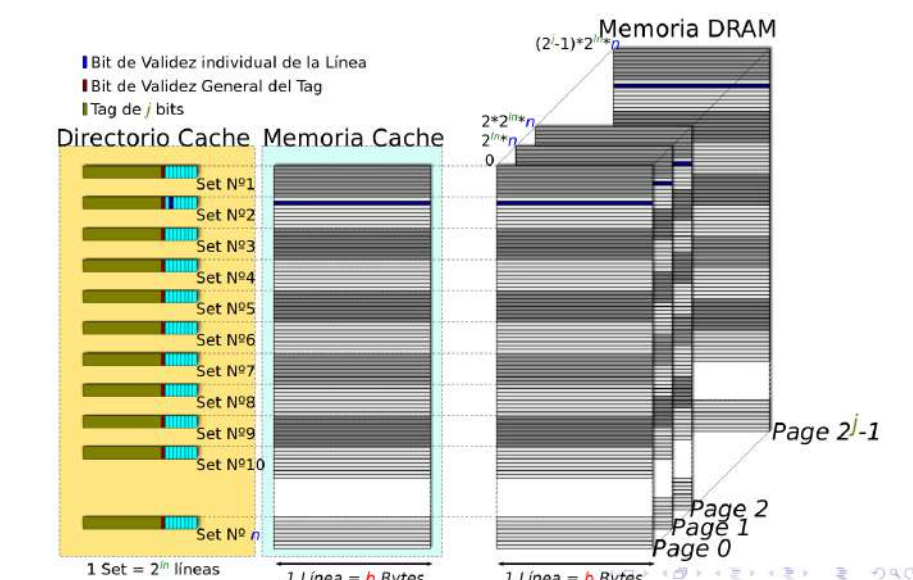


Figura 1.6: Sistema de Cache de Mapeo Directo

El controlador cache divide a la memoria principal en bancos (**páginas**), del mismo tamaño que la cache. Cuando las líneas de cache son muy pequeñas, se las suele agrupar en **conjuntos** (*sets*), típicamente de 8 líneas, de manera tal que en vez de traernos una sola línea, nos traemos un conjunto completo (principio de vecindad espacial).

En general, en los procesadores más modernos que cuentan con memorias cache obscuramente grandes, tenemos líneas de 64 bytes (un choclo de cosas), por lo que se trabaja directamente con líneas. Sin embargo, en algunos modelos de ARM se trabaja con conjuntos. En este caso, vamos a explicar el caso general de tener líneas y conjuntos.

Inicialmente, cuando prendemos el sistema, tenemos que activar la memoria cache. Esta va arrancar vacía (al ser una memoria volátil), por lo que todos los accesos iniciales van a dar *miss*. Como cada vez que tenemos un *miss* nos traemos el dato a la cache, el cache se va a ir llenando, aumentando el *hit rate*.

---

## 1.4. Políticas de reemplazo de contenido

Cuando estamos trabajando bajo el esquema de Mapeo Directo, si hay un conflicto por un espacio en la cache, solo hay que reemplazar la línea vieja por la nueva (vecindad temporal). No hay necesidad de ninguna política de reemplazo, ya que la ubicación del nuevo bloque es predeterminedada.

Sin embargo, cuando trabajamos con un esquema asociativo, ya sea completamente asociativo o asociativo por conjuntos de  $N$  vías, es necesario tener un algoritmo de reemplazo para determinar qué línea de la cache hay que reemplazar por la nueva. Para lograr una alta velocidad, dicho algoritmo debe implementarse en hardware.

Cuando trabajamos con un esquema completamente asociativo, se podría reemplazar cualquiera de las  $K$  líneas de la cache, mientras que con esquema de  $N$  vías tenemos solo  $N$  opciones.

### 1.4.1. Políticas de reemplazo de contenido

Se han probado varios algoritmos de reemplazo de contenido. Mencionamos cuatro de los más comunes.

#### Least Recently Used (LRU)

Podríamos suponer que es más probable que se vuelva a necesitar un dato que fue accedido recientemente con respecto a uno que no. Luego, la idea de LRU es realizar un seguimiento de la última vez que se accedió a cada línea de la cache, guardando el *timestamp* en el directorio, y remover aquella línea (o vía) que se haya utilizado menos recientemente.

Debido a que asumimos que es más probable que se haga referencia a ubicaciones de memoria utilizadas más recientemente, LRU debería dar el mejor hit-rate. LRU también es relativamente fácil de implementar para una cache completamente asociativa. El mecanismo de cache mantiene una lista separada de índices para todas las líneas de la cache. Cuando se hace referencia a una línea, se mueve al principio de la lista. Para el reemplazo, se usa la línea al final de la lista. Debido a su simplicidad de implementación, LRU es el algoritmo de reemplazo más popular.

#### Least Frequently Used (LFU)

La idea del esquema LFU consiste en que el sistema lleve un registro del número de veces que se hace referencia a una línea en la memoria. Cuando la cache está llena y requiere más espacio, el sistema removerá la línea (o vía) con la frecuencia de referencia más baja.

Este esquema se implementa asignando un contador a cada línea (o vía) que se carga en la cache. Cada vez que se hace una referencia a esa línea, el contador aumenta en uno. Cuando la cache está llena, y tiene una nueva línea que debe ser guardada, el sistema buscará la línea (o vía) con el contador más bajo, y lo reemplazará por la línea entrante, reiniciando el contador.

#### First In - First Out (FIFO)

El esquema de FIFO es otro enfoque popular. Con este algoritmo la línea (o vía) que ha estado en la cache por más tiempo sería la próxima en ser reemplazada.

#### Random

El problema con LRU y FIFO es que hay situaciones en las que pueden causar *trashing*, es decir que tiran constantemente una línea, luego la traen de vuelta, repetidamente. Algunas personas argumentan que el reemplazo *random*, aunque a veces arroja datos que se necesitarán pronto, nunca genera *trashing*. Desafortunadamente es difícil tener un reemplazo verdaderamente *random*, y además puede disminuir el rendimiento promedio.

A pesar de esto, los estudios de simulación han demostrado que el reemplazo randomizado proporciona solo un rendimiento ligeramente inferior a un algoritmo basado en el uso (LRU, LFU, FIFO).

---

## 1.5. Coherencia

### 1.5.1. Políticas de Escritura

Una variable que está en la cache también está alojada en alguna dirección de la memoria principal. Ambos valores deben ser consistentes. Idealmente, nos gustaría que ambas copias de la variable mantengan el mismo valor todo el tiempo. Frente a este problema hay varios modos de actuar, dependiendo de si el sistema dispone de una sola CPU o más de una. Estas alternativas de acción en las escrituras se conocen como **políticas de escritura** y constituyen una de las decisiones más importantes en el diseño del sistema de memoria.

#### Write through

El procesador realiza la escritura en la memoria principal y el controlador cache refresca el cache con el dato actualizado. Esto garantiza la coherencia entre ambos datos de manera absoluta, pero a cambio el tiempo para efectuar escrituras es penalizado con el tiempo de acceso a DRAM. Si bien la escritura requiere un acceso a la memoria principal, en aplicaciones reales, la mayoría de los accesos son de lectura, por lo que esta ralentización no es tan significativa. Además, en caso de que ocurra un error, sabemos que el estado de la memoria era válido.

#### Write through buffered

Es una opción mejorada del **write through**, ya que el procesador actualiza la cache y el controlador de la cache, eventualmente, actualiza la copia en la memoria DRAM (típicamente cuando ocurre un miss). Mientras tanto, el procesador continúa ejecutando usando los datos actualizados de la cache.

Para implementar esta política de escritura, el controlador cache debe disponer de un buffer de escrituras para encolar en él las operaciones de escritura a memoria. Notemos que es posible que el buffer de escritura se llene, ya que el procesador escribe sobre la memoria cache (rápida), mientras que el controlador tiene que escribir sobre la memoria DRAM (lenta). Por lo tanto, cuando se llene el buffer, el controlador va a tener que hacer esperar al procesador hasta que termine de copiar los datos.

#### Write Back o Copy back

La idea es esperar hasta el último momento para actualizar la DRAM, de manera tal que tenemos la copia en la cache completamente incoherente con el contenido de la DRAM. Se marcan por hardware las líneas de la memoria cache cuando el procesador escribe en ellas como "Dirty". Luego, al momento de eliminar esa línea de la cache, el controlador de la cache deberá actualizar el valor en memoria DRAM.

Desde el punto de vista de la coherencia, no se mantiene ningún tipo de coherencia. Sin embargo, desde el punto de vista de la performance, es un avión (el procesador siempre escribe en la cache). Notemos que si el procesador realiza un *miss* mientras que el controlador cache está accediendo a la DRAM para actualizar otro dato, se deberá esperar a que termine de actualizar el dato, para luego recibir del controlador de cache la habilitación de las líneas de control para acceder a la DRAM.

Otro problema de este esquema es que si un proceso falla antes de que se realice la escritura en la memoria principal, es posible que se pierdan los datos almacenados en la cache.

Ahora, analicemos esto primero en un espacio mono-procesador, y luego en un espacio multi-procesador. En un espacio mono-procesador, solo hay una única CPU en el sistema, por lo que la coherencia no resulta tan crítica. Luego, lo que nos va a importar más que nada sería la eficiencia de la política de escritura, por lo que ya podríamos ir descartando Write Through.

Luego nos quedan como candidatos Write Through Buffered y Copy Back. Si utilizamos Write Through Buffered, en principio, salvo que se realicen muchas escrituras consecutivas, el procesador estaría funcionando directamente con la cache. El problema está en que se estaría accediendo innecesariamente  $n$  veces al bus del sistema (para actualizar la copia en la memoria principal).

El problema con acceder  $n$  veces al bus del sistema, de forma innecesaria, es que el controlador cache que toma control del bus de sistema deshabilita a todos los demás controladores a que puedan manejar el bus de sistema. Esto se debe a que dos controladores no pueden acceder al mismo tiempo al mismo bus, ya que ponen eléctricamente dos salidas en funcionamiento sobre una única entrada. Si no se tiene este

cuidado, se tiene un efecto eléctrico catastrófico, quemando el circuito. Además, incluso si por casualidad el circuito está protegido y no se quema nada, aún así se produce un hecho de **contención de bus**, es decir, se mezcla toda la información y terminan metiendo basura por el bus, dando lugar a una ensalada de bytes.

Por lo tanto, si bien utilizar Write Through Buffered no empeora la performance del procesador que escribe sobre la variable, si empeora la performance de todo el resto del sistema, al acceder más veces al bus del sistema (este problema aumentaría aún más en sistemas con múltiples procesadores). Luego, podemos concluir que *Copy Back* nos resultaría la mejor opción.

Ahora consideremos un entorno como el que tenemos en la figura 1.7, en el que contamos con dos procesadores (pero podrían ser 4 u 8, las conclusiones son las mismas) y un único banco de memoria principal para ambos procesadores.

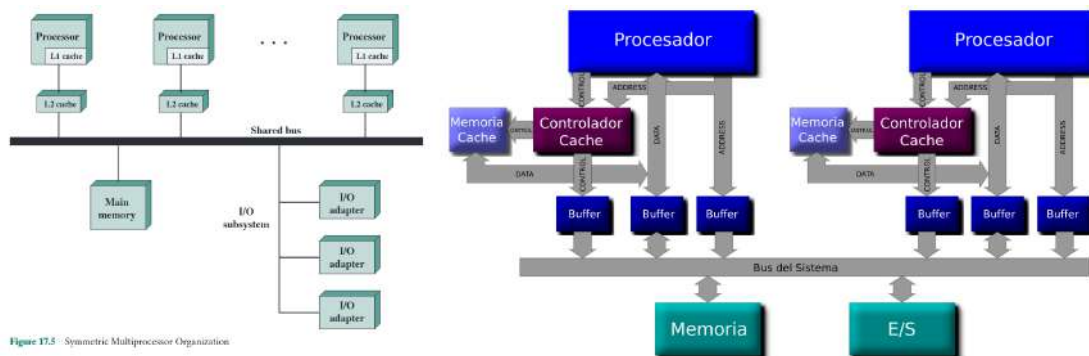


Figura 1.7: Coherencia en sistemas SMP

Aquí tenemos un escenario muy distinto al de un solo procesador, ya que podríamos tener una misma variable en las dos caches y en la memoria. Un caso típico de esto ocurre cuando programamos con *threads* (hilos de ejecución). Por ejemplo, si tenemos un *thread* de un *proceso* ejecutando en uno de los procesadores, y otro *thread* del mismo proceso ejecutando en el otro, ambos estarían compartiendo el mismo espacio de memoria (y variables compartidas). Por lo tanto, si uno cambia su valor, el otro se tiene que enterar, volviendo a la coherencia algo crítico.

En general, cuando trabajamos con multiprocesadores SMP, cada procesador tiene su propia cache. Por lo tanto, pueden existir múltiples copias de los mismos datos en diferentes caches simultáneamente. Luego, si los procesadores actualizaran sus copias libremente, esto podría resultar en una vista inconsistente de la memoria.

Esto se debe a que cuando uno de los dos procesadores modifica la copia, el único que tiene acceso al dato **actualizado** es el que escribió, y el resto solo tienen un dato inservible. Entonces, tenemos que actuar rápidamente para que el resto de los procesadores puedan enterarse cuanto antes del cambio, ya que si tuviesen esa misma variable en su cache, estarían trabajando con un dato incorrecto.

En principio, pareciera que Copy Back no puede usarse cuando hay más de un procesador, ya que el cambio no sería informado por el controlador cache ni siquiera a la DRAM, sino hasta que este dato se desaloje del cache (por falta de uso), pudiendo pasar que el otro thread utilice una copia mentirosa del dato. Más adelante veremos que esto no es tan así.

Ahora, suponiendo que usamos algunos de los métodos Write Through o Write Through Buffered, ¿cómo hacemos que el otro procesador se entere de que modificamos la variable compartida? Cuando trabajamos en un sistema SMP (Symmetric Multi-Processing), es decir con varios procesadores iguales, tiene que haber recursos de hardware disponibles para que se pueda *sincronizar* las variables, manteniéndolas coherentes.

Las soluciones a este problema se denominan generalmente **protocolos de coherencia** de cache. Estas soluciones proporcionan un reconocimiento dinámico en tiempo de ejecución de posibles condiciones

de inconsistencia, y resultan transparentes al desarrollo de software. Los distintos protocolos van a diferir en una serie de detalles, como por ejemplo dónde se almacena la información de estado en las líneas de datos, cómo se organiza esa información, dónde se aplica la coherencia y los mecanismos de ejecución, etc. En general, podemos dividir los protocolos de coherencia en dos categorías: **directory protocols** y **snoopy protocols**

### 1.5.2. Snoopy protocols

Los *snoopy protocols* distribuyen la responsabilidad de mantener la coherencia de la cache entre todos los controladores cache en un multiprocesador. Un cache debe poder reconocer cuándo sus líneas son o no compartidas. Cuando se quiere escribir sobre una línea de cache compartida, debe anunciarse a todas las demás caches mediante un mecanismo de *broadcast*. Cada controlador de cache debe ser capaz de "snopear" en la red para observar estas notificaciones transmitidas y reaccionar en consecuencia.

En particular, vamos a utilizar un bus adicional que se conecta a los controladores cache, conocido como **snoop bus** (bus que espía), el cual podemos ver en la figura 1.8. El snoop bus se encarga de traerle direcciones del bus del sistema al controlador cache.

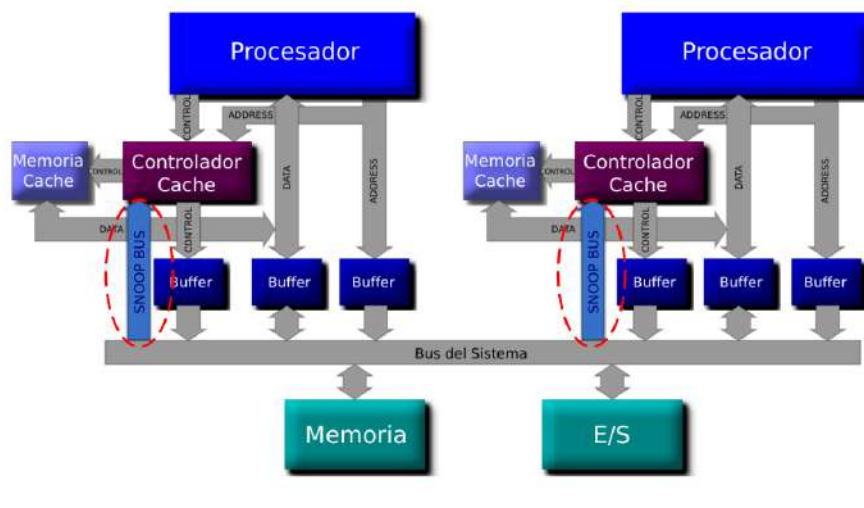


Figura 1.8: Snoopy bus

Cuando uno de los procesadores escribe sobre una variable en la memoria DRAM, el controlador cache necesita colocar en el bus de direcciones la dirección de la variable, activar por el bus de control un "Memory Write", y colocar en el bus de datos el valor que tiene que tener.

Entonces, el snoop bus lo que hace es capturar desde el bus del sistema la dirección de la variable que fue escrita, no le va a interesar qué valor tiene la variable, **solo le interesa su dirección**. Además, el snoop bus tiene las dos líneas de control que tienen que ver con la memoria: "Memory Read" y "Memory Write". Esto le permite saber qué direcciones están usando otros controladores y, además, qué es lo que van a hacer con esa dirección. A los efectos de la coherencia, si están leyendo la variable, no pasa nada. El problema está cuando la van a escribir.

Por lo tanto, el snoop bus **no es un bus entre los controladores**, sino que une a cada uno de los controladores de cache con el bus del sistema, pero lo hace en sentido inverso, de manera tal que el controlador cache pueda leer la dirección de la variable, permitiendo que cada controlador sepa qué están haciendo el resto de los procesadores con la memoria en cada momento.

Los snoopy protocols son ideales para un multiprocesador basado en único bus compartido, porque el bus proporciona un medio simple para broadcastear y snopear. Sin embargo, debido a que uno de los objetivos del uso de caches locales es evitar los accesos innecesarios al bus principal, se debe tener cuidado de que el aumento del tráfico del bus debido al broadcasteo y al snoqueo no anule las ganancias del uso de memorias cache.

---

Existen dos alternativas dentro de los snoopy protocols: **write invalidate** y **write update** (o write broadcast). Cuando trabajamos con un protocolo write-invalidate, podemos tener múltiples lectores pero solo un único escritor al mismo tiempo. Es decir, una línea puede ser compartida entre varias cache que solo estén realizando lecturas. Cuando una de las caches quiere escribir sobre la línea, primero debe mandar una señal que **invalida** esa línea en el resto de las cache, volviéndola exclusiva a la cache que quería escribir.

En cambio, cuando trabajamos con un protocolo **write-update**, pueden haber múltiples escritores y múltiples lectores al mismo tiempo. Cuando un procesador quiere escribir sobre una línea compartida, la variable a ser actualizada debe ser distribuida a todas las demás caches que la comparten (y así evitar tener incoherencias entre las distintas caches).

Ninguna de estas dos alternativas es mejor que la otra bajo todas las circunstancias. El rendimiento va a depender del número de caches locales y el patrón de lecturas y escrituras. Algunos sistemas implementan protocolos adaptativos que utilizan ambos mecanismos. Sin embargo, el mecanismo **write-invalidate** es el más ampliamente usado en sistemas comerciales multi-procesador, como es el caso de la familia x86. Por lo tanto, veamos más en detalle cómo funcionaría este tipo de protocolos.

### 1.5.3. Optimizaciones

Si bien utilizar el snoop bus resuelve el problema de la coherencia, hay lugar para algunas optimizaciones respecto a las políticas de escritura. Hasta ahora únicamente habíamos pensado en utilizar o bien Write Through o bien Write Through Buffer. Sin embargo, habíamos visto que, desde el punto de vista de la performance, Copy Back es el método más eficiente en cuanto a la cantidad de accesos al bus del sistema.

Sin embargo, pareciera un método inapropiado cuando se trata de mantener coherentes los datos entre dos o más caches. Luego, la idea va a ser utilizar Copy Back siempre que se pueda, y cuando no queda más remedio, utilizar Write Through o Write Though Buffered. Para esto, es necesario implementar un protocolo de coherencia que permita la comunicación entre los controladores de cache, para que, dependiendo del caso, se utilice Copy Back o Write Through, buscando una alta eficiencia, manteniendo la suficiente coherencia entre las distintas caches para el correcto funcionamiento de los programas.

Ahora veamos uno de los protocolos de coherencia más populares para el manejo de cache: el protocolo **M.E.S.I.**.

### 1.5.4. Protocolo M.E.S.I.

En el protocolo M.E.S.I., cada línea del cache puede tomar alguno de los siguientes estados

- **Modified**: La línea está presente *solo* en esta cache y sabemos que su contenido fue modificado respecto del valor en memoria (dirty). Se puede utilizar Copy Back hacia la memoria del sistema antes que otro procesador lea desde allí el dato (que no es válido).
- **Exclusive**: La línea está presente **solo** en esta cache y coincide con la copia en memoria principal (clean).
- **Shared**:
  - La línea del cache está presente, es consistente con la copia en memoria, y *puede* estar almacenada en caches de otros procesadores. Decimos "puede" porque el estado "Shared" es **impreciso**.
- **Invalid**: La línea de cache no es válida.

Al esquema anterior, se le agregan dos líneas "Shared" y "RFO" (**R**equest **F**or **O**wnership) para trabajar con el protocolo MESI. Ahora, vamos a ver los distintos escenarios en M.E.S.I., y cómo son las transiciones de un estado a otro.



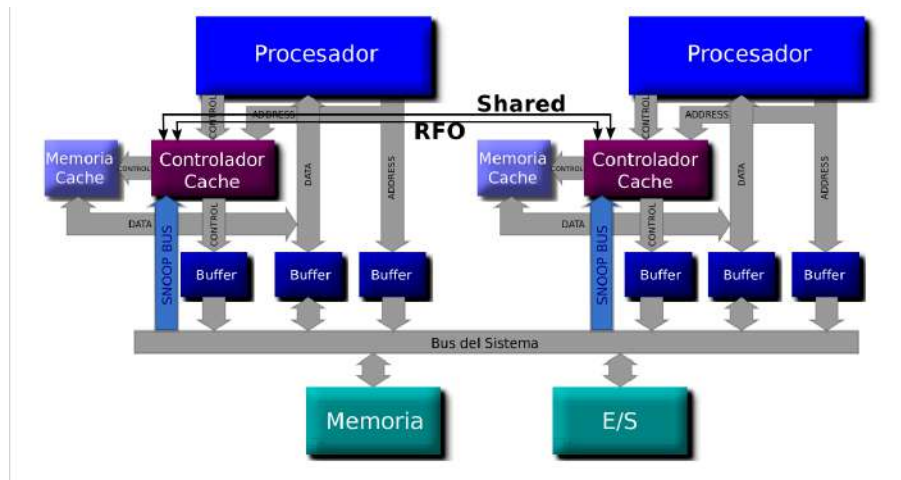


Figura 1.9: Coherencia en sistemas SMP con M.E.S.I.

Podemos resumir el significado de cada uno de los estados en la siguiente tabla:

**Table 17.1** MESI Cache Line States

	<b>M</b> Modified	<b>E</b> Exclusive	<b>S</b> Shared	<b>I</b> Invalid
This cache line valid?	Yes	Yes	Yes	No
The memory copy is ...	out of date	valid	valid	—
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line ...	does not go to bus	does not go to bus	goes to bus and updates cache	goes directly to bus

### Lectura:

- **Read Miss:** cuando tenemos un miss, ya sea porque no tenemos la línea cacheada o porque está marcada como Invalid, tenemos que iniciar una lectura a memoria. Cuando el controlador cache pida a la memoria traer la línea, el resto de los controladores revisan vía el snoop bus si tienen cacheado la línea.
  - Si una cache tiene esa línea en estado **Exclusive**, devuelve la señal "Shared" al cache lector indicando que esa línea es compartida. Luego, en ambos controladores se marca a esa línea como **Shared** en sus directorios de cache respectivos. Notemos que es importante mandar esta señal de "Shared" ya que, de otra manera, el controlador que pidió el ítem no tendría manera de enterarse si la línea es compartida o exclusiva.
  - Si una o más caches tienen una copia (limpia) en estado **Shared**, cada una de ellas envía una señal de "Shared" para notificar que la línea es compartida. El lector marca como **Shared** a la línea.
  - Si una cache tiene una copia en estado **Modified**, debe de algún modo "insertar" el dato mantenido en su línea, ya que este está incoherente con la memoria principal. La idea es que esa cache bloquea la lectura a memoria y provee esa línea al cache lector a través del bus compartido. Para ello, realiza la siguiente operatoria:
    1. Activa la línea "RFO" para indicar al lector que ese dato está incoherente (bloqueando la lectura a memoria y tomando control del bus compartido).
    2. Escribe en la memoria principal el valor actual de la línea.

- 
3. El lector copia ese valor actualizado a su cache cuando aparece por el bus de datos. (En algunas implementaciones, la cache con la línea modificada envía una señal al procesador lector para que reintente la lectura. Mientras tanto, el procesador con la copia modificada toma control del bus, actualiza la copia en memoria y marca su línea en estado **Shared**. Luego, el procesador lector intenta de nuevo y se encuentra que otro procesador tiene una copia limpia de su línea en estado **Shared**, y el resto se resuelve como en el caso anterior).
  4. Ambos pondrán esa línea en el estado **Shared**.
    - Si ninguna otra cache tiene una copia de esa línea, el lector se trae la línea de memoria y la marca como **Exclusive**.
  - **Read Hit**: Cuando intentamos leer una línea que está en la cache, el procesador simplemente lee la variable de la cache. No hay cambio de estado: el estado se mantiene **Modified**, **Shared**, o **Exclusive** (según corresponda).

#### Escritura:

- **Write Miss**: Cuando intentamos escribir sobre una línea que no se encuentra en la cache (o se encuentra marcada como Invalid), tenemos dos escenarios:
  - Si otra cache tiene una copia de la línea en estado **Modified**, tiene que alertar al escritor que tiene una copia modificada de la línea. Para ello, se realiza la siguiente operatoria:
    1. El otro controlador activa la línea "RFO" para indicar al escritor que ese dato está incoherente (bloqueando la lectura a memoria y tomando control del bus compartido).
    2. El otro controlador escribe en la memoria principal el valor actual de la línea, y marca como **Invalid** su propia copia (porque sabe que el escritor va a cambiar el contenido de esa línea).
    3. El escritor copia el valor actualizado a su cache cuando aparece en el bus de datos.
    4. Modifica su valor en la cache y la marca como **Modified**.
  - Caso contrario, el escritor se trae la línea y la modifica, pero debe enviar una señal de "RFO" para que si alguna otra cache tiene una copia sin modificar (**Shared** o **Exclusive**), invaliden su copia marcándola como **Invalid**.

Más tarde se programa una lectura del resto de la línea de cache y se marca como **Modified**.

- **Write Hit**: Cuando intentamos escribir una línea que está en la cache, tenemos que ver cuál es el estado actual de la línea:
  - **Shared**: Antes de actualizar el dato, el procesador debe obtener exclusividad sobre la línea. Para ello envía una señal de "RFO" para que cada cache que tenga una copia la marque como **Invalid**. El escritor entonces actualiza la copia y la marca como **Modified**.
  - **Exclusive**: El procesador ya tiene exclusividad sobre la línea, por lo que simplemente actualiza la copia y la marca como **Modified**.
  - **Modified**: El procesador ya tiene exclusividad de la línea y está marcada como **Modified**, por lo que solo tiene que actualizar su valor.
- Una línea que está en estado **Shared** o en estado **Exclusive** puede ser descartada (pasar a inválida) en cualquier momento, sin avisarle a nadie.
- Una línea que está en estado **Modified** también puede ser invalidada en cualquier momento, y tampoco le avisa al resto de los controladores. Sin embargo, se requiere actualizar previamente la memoria principal vía Copy Back.

Podemos ver en la figura 1.10 un diagrama de estados y transiciones del protocolo MESI.

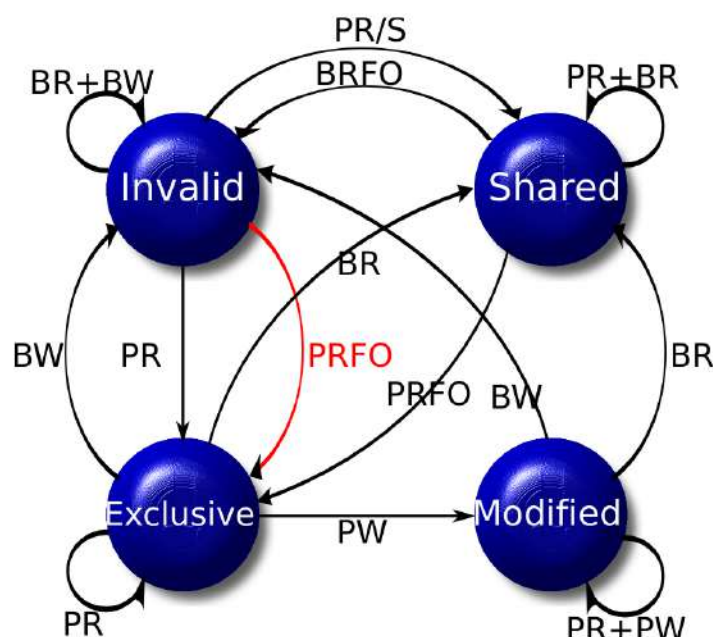


Figura 1.10: Diagrama de estados

- "PR" es *process read* (el procesador está leyendo).
- "BR" es *bus read* (se detectó una lectura de otro procesador).
- "BW" es *bus write* (se detectó una escritura de otro procesador).
- "PRFO" es *process request for ownership*.

Notemos que los estado **Modified** y **Exclusive** siempre son *precisos*, es decir, apuntan a una línea de cache que solo está en este cache, y por lo tanto señalan el *ownership* (qué controlador es dueño de esa línea).

Por otro lado, las líneas **Shared** son *imprecisas*. Esto se debe a que, pesar de que otros caches descarten esta línea, esta acción no se informa, y tampoco hay modo en que cada cache pueda llevar la cuenta de cuántos caches tienen la misma **Shared**. Por lo tanto, nunca puede pasar a **Exclusive**.

En este sentido, el estado **Exclusive** es el más apto para optimizar el mínimo de transacciones en el bus, ya que no requiere de informar nada al resto.

## 1.6. Arquitecturas de Cache

En algún momento de la evolución de los procesadores, la tecnología de integración alcanzó scalings que permitieron meter una cache (y el controlador cache) dentro del chip, a la que llamaremos cache de nivel 1, y una segunda cache afuera del chip.

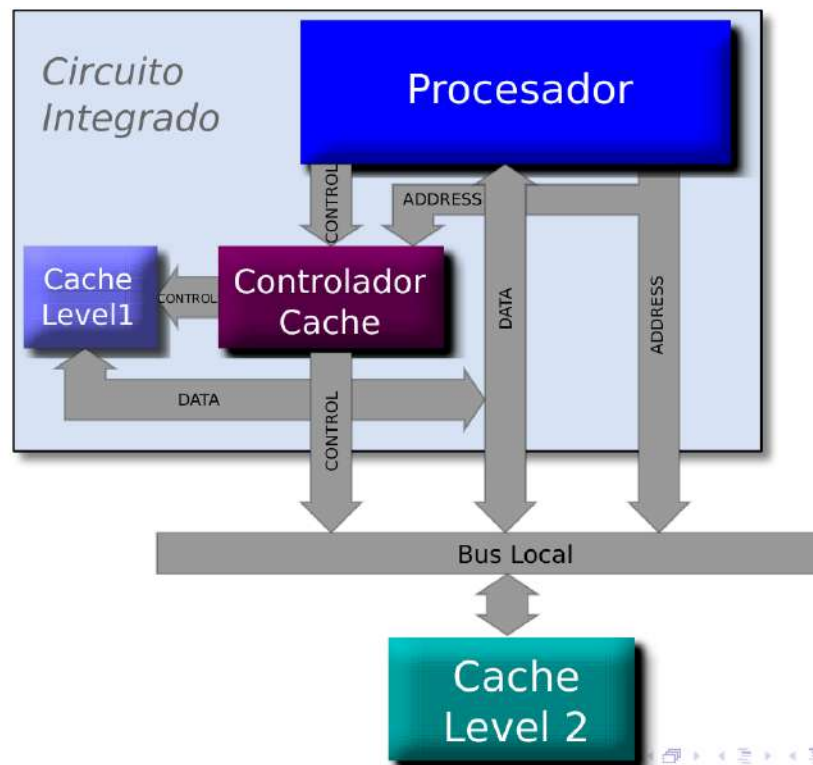


Figura 1.11: Cache Multinivel

La idea es tener una caché de nivel 1 que sea más chica, que trabaje a una mayor frecuencia y que esté más cerca del procesador. Además, al estar más cerca, se pueden utilizar buses más anchos, aumentando el ancho de banda.

A partir de la arquitectura Intel P6, la caché de nivel 2 empieza a estar dentro del chip. Si bien funcionaba más rápido que si estuviese fuera del chip, la caché de nivel 2 seguía trabajando a una frecuencia menor que la de nivel 1. Además, la caché de nivel 1 estaba desdoblada en una caché de datos y otra de código (microcódigo), lo cual permitía leer instrucciones y leer datos en el mismo ciclo de clock. Por último, teníamos una unidad de retiro que nos permitía almacenar resultados en el mismo ciclo de clock.

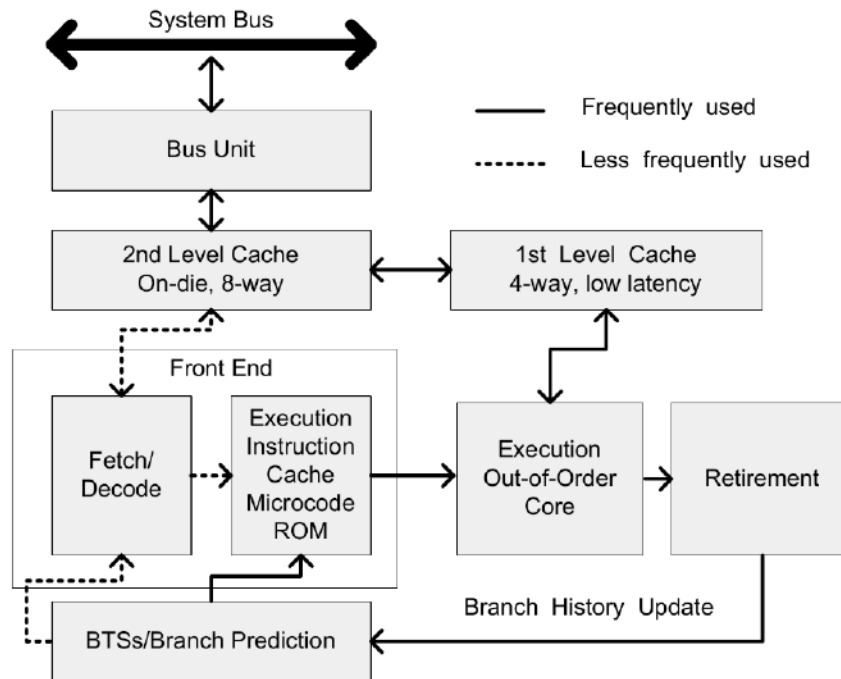


Figura 1.12: Arquitectura Intel P6

Luego, a partir de la arquitectura Intel Netburst, aparecieron las caches de nivel 3 opcionales, y actualmente tenemos arquitecturas como la Intel Skylake Server con tres niveles de cache.

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency <sup>1</sup>	Peak Bandwidth (bytes/cyc)	Sustained Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	96 (2x32B Load + 1x32B Store)	~81	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/4	64	12 cycle	64	~29	Writeback
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	<sup>44</sup>	32	~18	Writeback

Figura 1.13: Parámetros de Cache

Notación: /8 significa a 8 vías.

Típicamente, vamos a ver que el nivel de cache más alto es compartido: si se tiene L1 y L2, la L2 suele ser compartida; si se tiene L1, L2 y L3, la L3 suele ser compartida. Intel utiliza la tecnología *Smart Cache* para el manejo de estas memorias cache compartidas, aprovechando el mismo espacio para todos los cores. Esto nos sirve particularmente cuando trabajamos con aplicaciones que usan *threading*, ya que estas suelen compartir variables, ya que no tiene sentido tener varias copias de una variable compartida en distintas caches.

## Capítulo 2

# Intruction Level Parallelism (ILP)

A medida que la tecnología fue evolucionando y el costo del hardware fue disminuyendo, cada vez se fueron buscando más oportunidades para aumentar el paralelismo, generalmente, para mejorar el rendimiento, pero también para aumentar la disponibilidad de los recursos. El paralelismo se presenta en tres formas generales: paralelismo a nivel de instrucción, paralelismo a nivel de datos, y paralelismo a nivel de procesador.

En la materia se analizó paralelismo a nivel de datos cuando se vieron las extensiones SIMD (Single instruction, multiple data), que son recursos de la ISA para poder procesar muchos datos en una instrucción, y también hemos hablado de la necesidad de paralelizar buses, es decir, tener una arquitectura de tipo Harvard, para que el esfuerzo que hacemos dentro de la CPU no se termine atorando en la salida. En este capítulo vamos a analizar el paralelismo a nivel de instrucciones, y luego veremos algunos conceptos sobre paralelismo a nivel de procesador.

### 2.1. Pipeline

El primer punto que debemos analizar cuando hablamos de paralelismo a nivel de instrucciones es el **pipeline** de instrucciones. En la década de los 40, Von Neumann definió un modelo básico de cómputo, el cual fue referencia durante varias décadas. Algunos de esos conceptos ya fueron superados, pero otros como el concepto de máquina de ejecución son inevitables.

Sabemos que la ejecución automática involucra una cierta máquina de estado. Todos los procesadores tienen un modelo basado en etapas, algunos tienen más etapas, otros menos. Hasta 1977, la mayoría de los procesadores ejecutaban las distintas etapas en serie. Es decir, el procesador en el primer ciclo de clock buscaba la instrucción, en el segundo ciclo de clock la decodificaba, en el tercero buscaba los operandos, etc. En la figura 2.1 podemos observar una máquina de estados minimal de 5 etapas con ejecución secuencial.

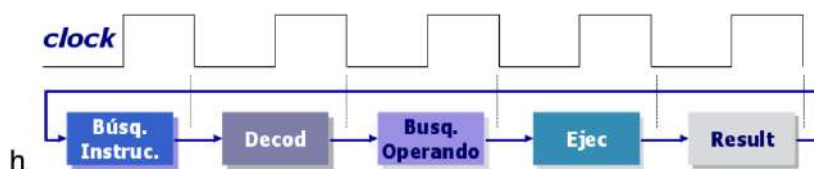


Figura 2.1: Etapas mínimas en la ejecución de una instrucción

Podemos ver que toda la actividad que hacía el procesador consistía en resolver una etapa particular a la vez, para luego pasar a la siguiente. Esto implicaba que todas las partes del procesador que no se utilizaban en una etapa particular quedaban en *stand-by*. Por ejemplo, cuando estábamos en la etapa de **Result** (guardar el resultado donde corresponda), la ALU no estaba haciendo nada.

Una de las utilidades que tenía este modelo era que se podía determinar qué recursos iban a estar siendo utilizados por la CPU mirando unos bits que nos decían en qué etapa de ejecución se encontraba la instrucción. Esto permitía que controladores externos que necesitaban utilizar algún recurso, por ejemplo, el bus para hacer un acceso a memoria, podían usar el bus en los ciclos de clock en los que el procesador se encontraba en una etapa que no use el bus.

Si bien este mecanismo tenía estas pequeñas ventajas y funcionó bien en las primeras generaciones de procesadores, traía consigo grandes desventajas. Luego, en la medida en que aumentaron las necesidades de tiempos de ejecución más cortos, este modelo fue revisado.

Desde hace años se sabe que la búsqueda de instrucciones a memoria es un cuello de botella importante en la velocidad de ejecución de las instrucciones. Para aliviar este problema, lo primero que se planteó fue tener la capacidad de obtener instrucciones de la memoria por adelantado (**pre-fetching**), de manera tal que estuvieran disponibles para cuando fueran necesarias.

En cierto sentido, cuando hacemos pre-fetching estamos dividiendo la ejecución de la instrucción en dos partes independientes: la obtención de instrucciones y la ejecución de las mismas. El concepto de **pipeline** lleva esta estrategia mucho más allá. En lugar de dividirse en solo dos partes, la ejecución de instrucciones se divide en muchas partes (a menudo una docena o más), donde cada etapa es manejada por una unidad dedicada.

Las arquitecturas con **pipeline** permiten superponer en el tiempo el procesamiento de varias instrucciones a la vez. Solo se necesita que los bloques del procesador que resuelven los distintos estados, operen de forma simultánea. El concepto clave es que todos los bloques trabajen en paralelo, pero cada uno en una instrucción **diferente**.

Esta idea está inspirada en el concepto de una línea de montaje, en donde cada operación se descompone en partes, y diferentes partes se ejecutan al mismo tiempo. En la figura 2.2 podemos observar un modelo teórico e ideal de un pipeline.

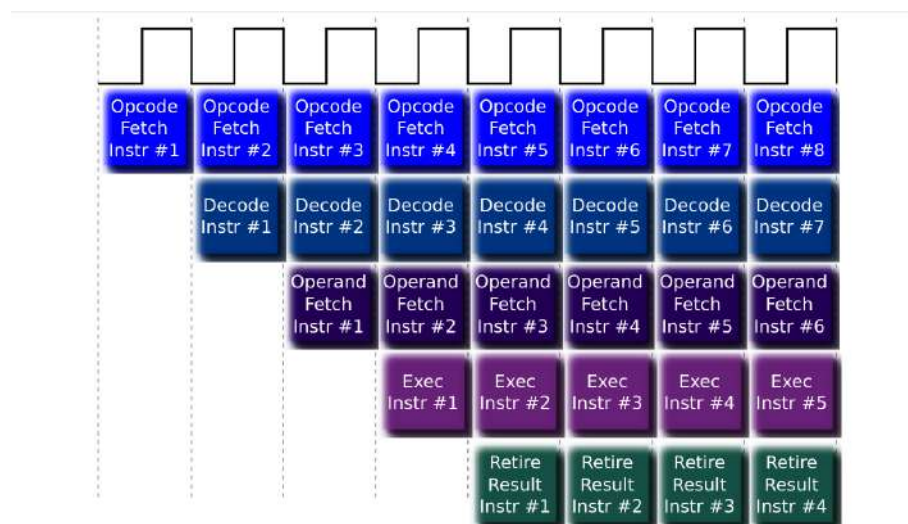


Figura 2.2: Situación Ideal de un Pipeline

Cuando encendemos la máquina, la primera instrucción se fetchea en el primer ciclo de clock. En el segundo ciclo de clock, se decodifica la primera instrucción, y como la unidad de **Fetch** ya terminó de fetchear la instrucción 1, puede fetchear la instrucción 2. Las demás unidades no tienen nada que hacer todavía, por lo que todavía están en *stand-by*. A medida que van pasando los ciclos de clock, se van ocupando el resto de las unidades.

Si seguimos así, en 5 ciclos de clocks luego de haber iniciado la máquina, habremos obtenido el primer resultado. Hasta acá, no tenemos ninguna diferencia respecto del modelo de ejecución en serie, ya que ambos obtuvieron el primer resultado en 5 ciclos de clock. Sin embargo, la diferencia está en que, en

---

el modelo de ejecución en serie, el segundo resultado se obtenía en el ciclo de clock número 10, mientras que en el modelo de pipeline, el segundo resultado se obtiene en el ciclo de clock 6, el tercero en el 7, el cuarto en el 8, etc.

Introducida la ejecución en pipeline, pensemos ahora qué relación tiene la cantidad de etapas en una ejecución con el tiempo de ejecución de instrucciones. Si pensamos en el modelo de ejecución en serie, resulta indeseable tener muchas etapas, porque si cada una nos consume un ciclo de clock y seguimos agregando etapas, por más que cada etapa sea más simple, la ejecución se vuelve cada vez más lenta, debido a que se introduce un overhead asociado al hecho de tener que coordinar las distintas etapas entre sí.

Ahora, en el caso del modelo de ejecución en pipeline, pareciera que cuanto más etapas tenemos, si bien se tarda más en completar cada una de las instrucciones por separado, es más factible que todas las etapas se puedan ejecutar en un ciclo de clock, obteniéndose una tasa de una instrucción por ciclo de clock.

Podemos relacionar (de forma teórica) el tiempo de procesamiento interno de una instrucción para una arquitectura de ejecución en serie con el tiempo que llevaría procesarla en un sistema con pipeline, mediante la siguiente fórmula

$$TimePerInstruction = \frac{\text{Tiempo de ejecución en serie}}{\text{Cantidad de etapas}}$$

Esta fórmula nos estaría diciendo que a mayor cantidad de etapas, el tiempo de procesamiento por cada instrucción disminuye.

Esto nos estaría diciendo que, si bien el pipeline no reduce el tiempo de ejecución de cada instrucción individual (**latencia**), al aplicarse en paralelo al flujo de instrucciones, incrementa la cantidad de resultados por unidad de tiempo (**throughput**).

Podría parecer que cuanto mayor sea el número de etapas en el pipeline, más rápida será la tasa de ejecución. Sin embargo, podemos señalar algunos factores que frustran este patrón aparentemente simple:

- En cada etapa del pipeline, tenemos un cierto overhead asociado a mover los datos de un buffer a otro. Este overhead puede aumentar de forma apreciable el tiempo de ejecución de cada instrucción.
- La cantidad de lógica de control requerida para el manejo de dependencias (lo veremos más adelante) y para optimizar el uso del pipeline aumenta enormemente con la cantidad de etapas.

### 2.1.1. Obstáculos y Dependencias

Una vez visto el modelo teórico, veamos cómo funcionan los pipelines ajustándonos un poco más a la práctica. Se le llaman *hazards* (obstáculos) a todo lo que conspira contra la eficiencia de un pipeline, es decir todo lo que hace que el pipeline en la práctica no sea como el teórico. Podemos agrupar a estos obstáculos en 3 categorías:

- Obstáculos estructurales.
- Obstáculos de datos:
  - RAW
  - WAR
  - WAW
- Obstáculos de control.

En general, cuando no podemos evitar alguno de estos obstáculos, el efecto que se produce se lo denomina **pipeline stall** (congestión del pipeline). El pipeline debe detenerse porque tenemos ciertas



---

condiciones que no permiten la ejecución de **alguna** instrucción, degradando la performance del procesador respecto del comportamiento ideal del pipeline.

### Obstáculos Estructurales:

Un **obstáculo estructural** ocurre cuando dos (o más) instrucciones compiten por el mismo recurso al mismo tiempo. Ejemplos de recursos incluyen memorias, caches, buses y unidades funcionales. Este tipo de obstáculos se pueden dar por varias razones:

- Una etapa no está lo suficientemente atomizada, es decir que aún concentra demasiadas funciones, por lo que para completarse requiere de más de un ciclo de clock. En principio, podríamos intentar dividir esta etapa en algunas etapas más pequeñas, pero esto no siempre es posible (hay que ver qué se puede dividir, si hay lógica suficiente y analizar qué problemas puede traer).
- Si dos instrucciones que van a utilizar una etapa están a una distancia temporal menor al tiempo que la etapa consume para procesar, estas instrucciones van a estar en un conflicto de recursos de ejecución.

Por ejemplo, podría ocurrir que haya dos instrucciones que utilicen la FPU y estén a una distancia temporal tal que cuando la segunda quiera ingresar a la etapa que le corresponde para su ejecución, va a encontrar que la unidad de ejecución va a estar ocupada con la anterior, por lo que va a tener que esperar a que la primera termine.

Veamos un ejemplo para entender un poco mejor este tipo de obstáculos. Consideremos un procesador que solo tiene una etapa que permite acceder a memoria, tiene un solo bus para acceder a memoria, una sola unidad de generación de direcciones de memoria, y esa unidad está compartida tanto para datos como instrucciones. Entonces, en el caso en el que se necesite un operando de memoria, el acceso para traer ese operando podría interferir con otra búsqueda de operando, e incluso con el **Fetch** de la siguiente instrucción.

Si volvemos al ejemplo de la figura 2.2, pero ahora tomamos en cuenta los obstáculos estructurales en este esquema, debemos corregir la ejecución del pipeline para resolver los conflictos de los recursos. Para ello, cada vez que tengamos un conflicto, vamos a resolver dándole prioridad a aquella instrucción que se encuentra más avanzada en su ejecución:

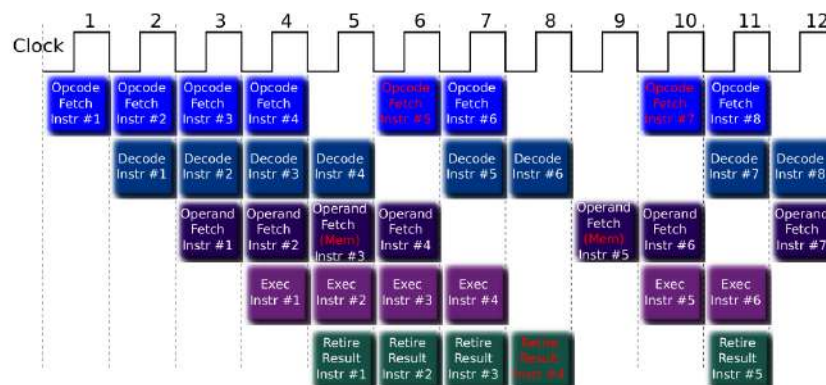


Figura 2.3: Obstáculos Estructurales

Notemos que en la figura 2.2, fetchear la instrucción 8 estaba previsto en el ciclo de instrucción 8, pero termina pasando en el ciclo de instrucción 11 (3 ciclos de demora). A pesar de estos problemas, sigue siendo mucho más eficiente que el modelo de ejecución en serie, que hubiese tardado 35 ciclos de clock.

Para reducir este tipo de obstáculos, la solución más sencilla es agregar hardware. Por ejemplo, se pueden utilizar algunas de las siguientes opciones (por separado o combinadas):

- Desdoblar la L1 cache en datos e instrucciones. Esto es un gran acierto, especialmente en procesadores que permiten utilizar a la memoria como operandos y destino. El desdoblamiento de la cache, con buses separados y unidades de generación de direcciones separadas es un golazo, ya que todos los problemas que tuvimos en la figura 2.3 no existirían.
- Utilizar buffers de instrucciones, implementadas como pequeñas colas FIFO para hacer pre-fetching, de manera tal que si se tiene acceso al bus, la unidad de **Fetch** vaya trayendo las próximas instrucciones, almacenándolas en el buffer FIFO.
- Ensanchar de los buses más allá del ancho de palabra del procesador. De esta manera, podemos llegar a fetchear varias instrucciones de una o traernos muchos operandos de una (principio de vecindad). Esta resulta la solución más práctica porque ensanchar los buses no resulta demasiado costoso y presta beneficios realmente importantes.
- También podemos aumentar la profundidad del pipeline (sin exagerar), lo cual produce etapas mucho más simples, y permiten asegurar que cada tarea se pueda resolver en un ciclo de clock.

### Obstáculos de datos:

Este tipo de obstáculos se producen cuando, por efectos del pipeline, dos instrucciones que acceden a un mismo **dato**, se ejecutan de manera tal se obtenga un resultado diferente al que ocurriría con una ejecución secuencial.

En general, podemos dividir a este tipo de obstáculos en tres tipos. Para ello, consideremos dos instrucciones *I1* e *I2*, con *I1* precediendo a *I2* en el programa.

- **Read After Write (RAW)**: *I1* quiere modificar un registro o ubicación de memoria e *I2* lee los datos en esa memoria o ubicación de registro. Se produce un obstáculo de tipo RAW si la lectura tiene lugar antes de que se complete la operación de escritura. Este tipo de obstáculo es el más común y corresponde a una dependencia de datos verdadera. Para resolver este tipo de obstáculos, es necesario que el orden del programa sea preservado, y así asegurar que *I2* reciba el valor escrito por *I1*.
- **Write After Read (WAR)**: *I2* quiere escribir sobre un registro (o variable en memoria) e *I1* quiere leer ese mismo dato. Se produce un obstáculo de tipo WAR si *I2* escribe el dato antes de que *I1* pueda leerlo. Este tipo de obstáculos corresponde a una antidependencia, y pueden ocurrir cuando algunas instrucciones que escriben el resultado en una etapa temprana del pipeline y otra instrucción lee en una etapa tardía del pipeline, o bien cuando las instrucciones son reordenadas.
- **Write After Write (WAW)**: *I2* quiere escribir sobre un registro e *I1* quiere escribir ese mismo registro. Se produce un obstáculo de tipo WAW si *I2* termina escribiendo sobre el registro antes de que *I1* pueda hacerlo, dejando como resultado el valor escrito por *I1* en vez del escrito por *I2*. Este tipo de obstáculos corresponde a una dependencia de salida, y está presente solo en los pipelines que permiten escrituras en más de una etapa o o bien cuando las instrucciones son reordenadas.

Por ahora vamos a centrarnos en los obstáculos de tipo RAW (cuando veamos ejecución fuera de orden vamos a analizar WAR y WAW). Consideremos el siguiente código para un procesador MIPS:

```

1  add R1, R2, R3
2  sub R4, R1, R5
3  and R6, R1, R7
4  or  R8, R1, R9
5  xor R10, R1, R11

```

Listing 2.1: Obstáculos de datos

La primera instrucción suma los registros *R2* y *R3*, y su resultado lo guarda en *R1*. La segunda instrucción resta al registro *R1* con *R5*, y su el resultado lo guarda en *R4*. El registro *R1*, en la primer instrucción es el destino y en la segunda es uno de los operandos. Por lo tanto, tenemos un obstáculo de tipo RAW, y hasta que no se termine de ejecutar la instrucción 1, la instrucción 2 no tiene sus operandos válidos,

ya que  $R1$  no tiene un valor válido.

Entonces, ¿qué pasa con el pipeline?

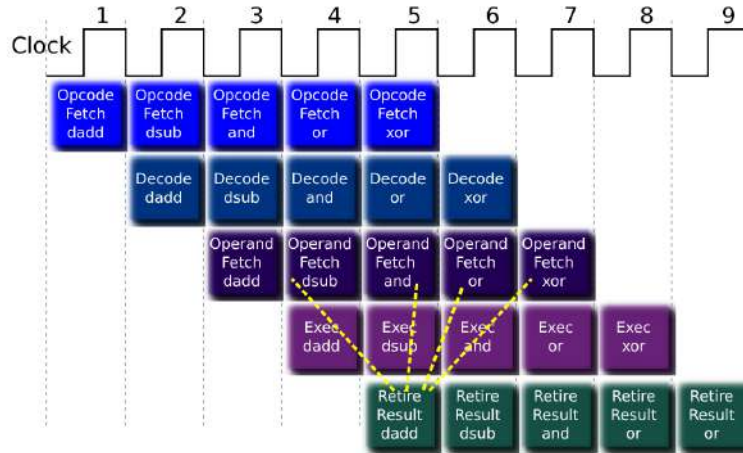


Figura 2.4: Obstáculos de Datos - Conflicto de recursos

En el 5<sup>to</sup> ciclo de clock, se genera el resultado de la suma ( $R1$  válido), pero  $R1$  es requerido como operando en las cuatro instrucciones restantes. En el caso de OR y XOR no tendrían problemas porque los operandos se buscan luego de la fase de resultado de la instrucción ADD. Sin embargo, la instrucción SUB y la instrucción AND no pueden conseguir a tiempo el resultado de la suma, para poder usarlo como operando. Entonces, tenemos que hacer que estas instrucciones esperen a que se obtenga el resultado.

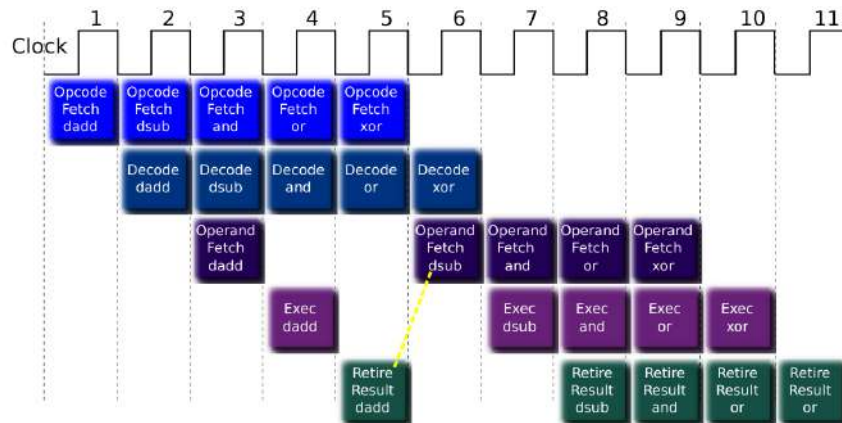


Figura 2.5: Obstáculos de Datos - Efecto en CPI

En la figura 2.5 podemos observar el efecto que genera tener estas dependencias de datos. Ahora, algo que hacíamos en 9 ciclos, tardamos 11. En general,  $CPI = CPI + n$ , siendo  $n$  la distancia entre las etapas del pipeline que dependen entre sí.

¿Cómo podemos reducir este efecto? Una forma de reducir este efecto es el *Forwarding*. Cuando estamos ejecutando el ADD en el 4<sup>to</sup> ciclo de clock, el resultado de la suma va a parar al registro de resultado de la ALU (ALUOut). Luego, en el 5<sup>to</sup> ciclo de clock, el resultado se copia internamente el valor en ALUOut en el registro destino, en este caso  $R1$ .

Ahora, si en el 4<sup>to</sup> ciclo de clock, una vez tengamos el resultado, mediante una lógica muy sencilla, realimentamos el valor de ALUOut en alguno de los registros de operandos de la ALU, no deberíamos

tener ningún inconveniente. Entonces, en vez de esperar a ponerlo en  $R1$ , para que después  $R1$  lo escriba en la ALU devuelta, la idea va a ser dejarlo disponible para  $R1$ , pero además escribirlo devuelta en la ALU.

En definitiva, lo que estamos haciendo es retroalimentar (*forwarding*) el operando de la ALU (FPU, o la unidad que corresponda) con el resultado obtenido en ALUOut. Esto permite disponer del dato en la siguiente instrucción, ahorrando el tiempo de escritura en el operando destino.

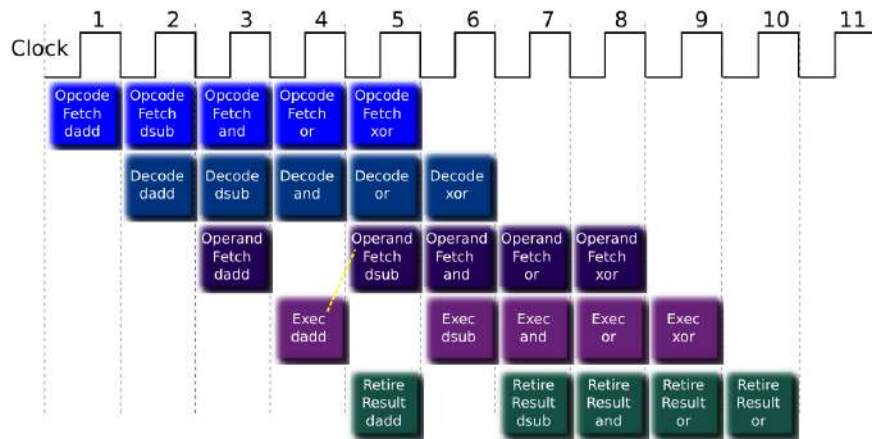


Figura 2.6: Obstáculos de datos - Forwarding

Notemos que solo es posible hacer *Forwarding* para un cierto subconjunto de operaciones de ALU, denominadas *back-to-back*. En cambio, en operaciones como LOAD no es posible hacer *forwarding*.

```

1  ld  R1, 0(R2)
2  sub R4, R1, R5
3  and R6, R1, R7
4  or  R8, R1, R9
5  xor R10, R1, R11

```

En este caso,  $R1 \leftarrow Mem(R2+0)$ , y como el resultado no pasa por la ALU, no podemos retroalimentarlo a alguno de los registros operando de la ALU.

### Obstáculos de Control:

El tercer tipo de obstáculos son los **obstáculos de control**, correspondientes a una dependencia de control. Una dependencia de control determina el orden entre una instrucción  $I$  con respecto a una instrucción de *branch*, de manera tal que la instrucción  $I$  sea ejecutada respetando la lógica del programa.

Un *branch* es la peor situación en pérdida de performance, ya que tenemos una discontinuidad en el flujo de ejecución, pero el pipeline trabaja con instrucciones en secuencia. Por lo tanto, el funcionamiento de un *branch* resulta contradictorio al funcionamiento de un pipeline.

En general, las dependencias de control nos imponen dos restricciones:

1. Una instrucción que sea control dependiente a un branch (que se ejecute dependiendo de la evaluación del branch) no puede ser movida antes del branch, provocando que ya no sea controlada por el branch.
2. Una instrucción que no sea control dependiente en un branch no puede ser movida después del branch, provocando que su ejecución sea controlada por el branch.

Es decir, las instrucciones próximas a un *branch* terminan siendo dependientes al branch, y no pueden ejecutarse hasta que la condición del branch sea evaluada (más adelante vamos a ver cómo

podemos esquivar esto). Luego, para corregir la ejecución del programa, una parte de lo que estaba pre-procesado debe descartarse, y retomar la ejecución desde la instrucción que corresponda. Este efecto se conoce como **branch penalty**.

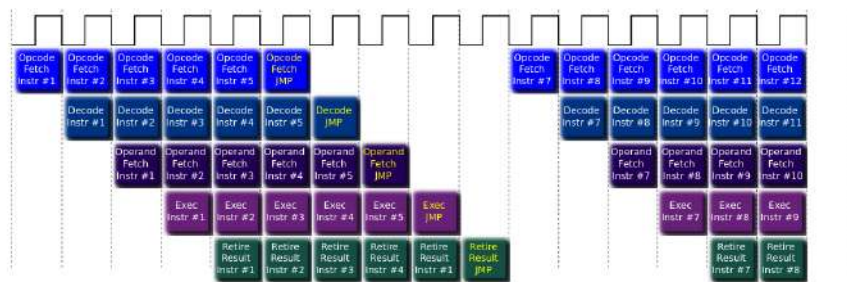


Figura 2.7: "La conspiración de los branches"

Notemos que es aquí donde tener demasiadas etapas nos juega en contra, justamente porque el *branch penalty* depende directamente de la cantidad de etapas en el pipeline. La situación con el manejo de interrupciones es la misma.

Típicamente, para poder corregir la ejecución del pipeline vamos a tener que esperar a que la instrucción de branch llegue a una etapa determinada, dependiendo del tipo de salto que sea:

- Si es un salto incondicional (o una llamada a subrutina) con direccionamiento directo, vamos a poder corregir la ejecución en la etapa de decodificación.
- Si es un salto incondicional (o llamada a subrutina) con direccionamiento indirecto, en la etapa de búsqueda de operando.
- Si es un salto condicional, en la etapa de ejecución.

Por último, si bien *Forwarding* puede ayudar a disminuir el efecto de los diferentes casos expuestos anteriormente, no es una solución óptima, ya que solo lograremos disminuir el **branch penalty** en algunos pocos ciclos de clock. Para soluciones más eficientes, es necesario recurrir a algoritmos un poco más sofisticados. Entramos, entonces, al universo de las **unidades de predicción de saltos**.

## 2.2. Unidades de Predicción de Saltos

En este caso, si ignoramos la dependencia de control y movemos  $y = 1/x$  antes del branch, se podría generar una excepción. Notemos que este código se podría traducir en lenguaje de máquina de manera tal que no tengamos ninguna dependencia de datos, por lo que lo único que nos previene de intercambiar estas dos instrucciones es la dependencia de control.

¿Cuál es la necesidad de predecir los saltos? Cuando trabajábamos con un pipeline sencillo de 5 etapas, los branch penalties no parecían tan grave. Sin embargo, a medida que aumenta la complejidad de un procesador, con pipelines más profundos y mayor paralelización, los branch penalties aumenta de forma proporcional, medidos en ciclos de clock y en cantidad de instrucciones por ciclo de clock. Por lo tanto, es crucial lograr una correcta predicción de saltos para reducir las **branch penalties**.

Hasta ahora, la manera en la que resolvíamos los obstáculos de control generados por los branches era descartar las etapas de las instrucciones pre-procesadas y volver a fetchear las instrucciones posteriores al resultado del branch, para luego continuar la ejecución normalmente. La idea, entonces, va a ser generar una predicción, ya sea estática o dinámica, que nos diga si el branch será taken o non-taken, de manera tal de que, una vez sepamos a qué dirección saltar, podamos continuar con el fetcheo de instrucciones. Si la predicción termina siendo errónea, debemos descartar lo pre-procesador, y retomar desde donde corresponda.

---

### 2.2.1. Predicción de saltos estática

Las técnicas estáticas de predicción de saltos se basan en que el compilador identifique las estructuras de los lenguajes de alto nivel como **for**, **if**, etc, y en base a esas estructuras utilizadas decida si va a tomar el salto o no. Algunas de estas técnicas requieren de cambios en la ISA, como agregar un segundo conjunto de instrucciones de branches condicionales tales que contengan un bit que el compilador pueda modificar para indicar su predicción del branch.

#### Predicted-non-taken

Para empezar, podemos probar asumiendo que el salto **nunca** se toma. Es decir, siempre vamos a continuar la ejecución como si el salto fuese una instrucción como cualquier otra. En general, este modelo podría funcionar adecuadamente en estructuras de programa como la siguiente:

1. instrucción branch.
2. instrucción sucesora secuencial.
3. instrucción target del branch taken.

donde la **sucesora secuencial** es la instrucción siguiente en secuencia a la del branch, mientras que la instrucción **target** es la instrucción a la cual se salta. Notemos que esta estructura no es otra cosa que hacer un *if*. Si el branch resulta **taken**, las instrucciones que fueron fetcheadas y decodificadas deben ser descartadas, y la ejecución se retoma en la dirección destino.

#### Predicted-taken

Ahora, el procesador asume que el salto se toma siempre y, por lo tanto, continúa la ejecución de las instrucciones a partir de la dirección *target*. En general, este modelo funciona adecuadamente en estructuras de programa como la siguiente:

1. instrucción target del branch taken.
2. grupo de instrucciones a ejecutar iterativamente.
3. instrucción branch.

Esta estructura no es otra cosa que un *loop*, por lo que este modelo funciona bárbaro en los ciclos *for* y en los ciclos *while*.

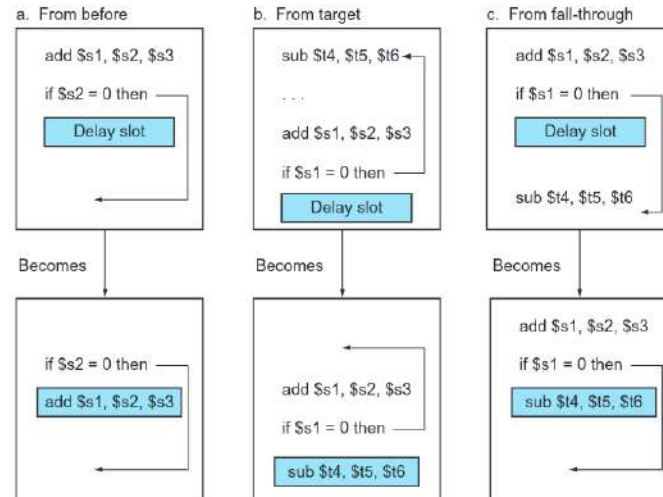
#### Delayed branch

En los primeros procesadores RISC, se introdujo un enfoque superior al *non-taken*, denominado **delayed branch**. Esto permite que el procesador evalúe la condición de branch antes de que se hayan pre-fetchead instrucciones inválidas. Con este método, el procesador siempre ejecuta la única instrucción que sigue inmediatamente al branch. Esto mantiene el pipeline lleno mientras el procesador busca un nuevo flujo de instrucciones.

1. instrucción branch.
2. instrucción sucesora secuencial.
3. instrucción target del branch taken.

La idea es ejecutar tanto la *target* como la sucesora secuencial siempre, y guardarse el resultado de la instrucción sucesora en un slot dedicado conocido como **delayed branch**. Luego, una vez se conoce el resultado de la evaluación de la condición, se aplica el resultado que corresponda. Esto mantiene el pipeline lleno para estas estructuras con una sola instrucción en el medio, lo que resolvió buena parte de los *if*.





**FIGURE 4.64 Scheduling the branch delay slot.** The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of `$s1` in the branch condition prevents the `add` instruction (whose destination is `$s1`) from being moved into the branch delay slot. In (b) the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the `sub` instruction when the branch goes in the unexpected direction. By “OK” we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, if `$t4` were an unused temporary register when the branch goes in the unexpected direction.

Sin embargo, con el desarrollo de las máquinas superescalares, esta estrategia requiere de poder ejecutar múltiples instrucciones y guardarlas en el delayed slot, generando varios problemas asociados a la dependencia entre instrucciones.

## Loop Unrolling

La idea es que el compilador desenrolle los branches, pero es necesario que los datos dentro del *loop* sean paralelizables. Esto nos permite reducir el overhead asociado al loop, ya que reducimos la cantidad de veces en las que se pregunta por la condición del loop. Además, se puede aumentar el paralelismo a nivel de instrucciones y mejorar la localidad de los registros, cache de datos, etc. La figura 2.2.1 ilustra estas tres mejoras en un ejemplo. El paralelismo de instrucciones aumenta porque la segunda asignación se puede realizar mientras se almacenan los resultados de la primera y se actualizan las variables del loop. Si los elementos del arreglo se asignan a los registros, la localidad del registro mejorará porque a `[i]` y a `[i + 1]` se usan dos veces en el cuerpo del loop, reduciendo el número de loads por cada iteración de tres a dos.

```

do i=2, n-1
    a[i] = a[i] + a[i-1] * a[i+1]
end do

```

(a) Original loop

```

do i=2, n-2, 2
    a[i] = a[i] + a[i-1] * a[i+1]
    a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2, 2) = 1) then
    a[n-1] = a[n-1] + a[n-2] * a[n]
end if

```

(b) Loop unrolled twice

Figure 15.8 Loop Unrolling

### 2.2.2. Predicción de saltos dinámica

Todos los métodos vistos hasta aquí dependen exclusivamente del compilador y de la ISA. Esto significa que el hardware interno del procesador no realiza ningún análisis del código ni agrega adaptatividad. Ahora, el siguiente paso es que el procesador comience a efectuar análisis de flujo en las instrucciones, y que tome decisiones en base a lo que va viendo adelante de la instrucción de salto. Los métodos siguientes son de **predicción dinámica** y corresponden a arquitecturas más avanzadas (Intel, ARM modernos), con mayor paralelismo a nivel de instrucciones.

#### Branch Prediction Buffer

El esquema más básico se conoce como **Branch Prediction Buffer**. Este consiste en una tabla, una memoria cache pequeña adentro del procesador, indexada por los bits menos significativos de la dirección de memoria de la instrucción de salto, donde se guarda un bit que representa el último resultado obtenido (0 = *non-taken*, 1 = *taken*).

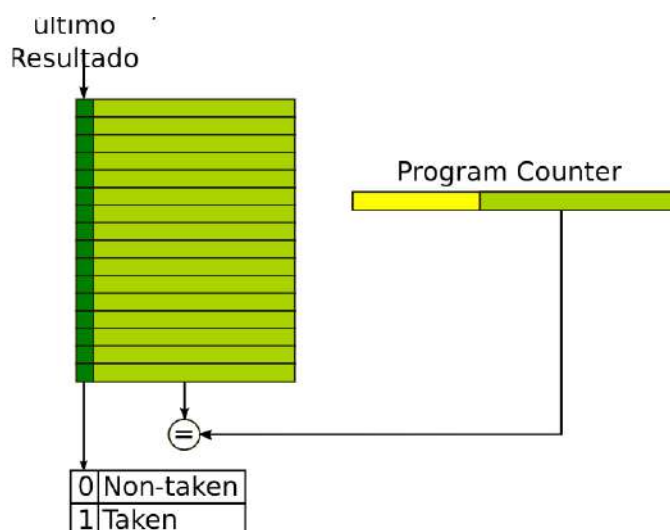


Figura 2.8: Branch Prediction Buffer - 1 bit

Luego, si el valor era 0, no saltamos; y si era 1, saltamos. Una vez que se conoce la evaluación de la condición, se refresca este bit con el valor del salto. Notemos que, como estamos usando solo una parte



de la dirección, es posible que la predicción que estemos tomando corresponda a otro salto. Sin embargo, esto no afecta a la correctitud del esquema, ya que solo se trata de una predicción que esperamos que sea correcta, para ir fetcheando a partir de la dirección predicha. Si la predicción termina siendo errónea, las instrucciones pre-procesadas deben ser descartadas, y el bit de predicción es invertido, para luego volver hacia la secuencia correspondiente.

El esquema de predicción de un bit tiene un defecto de rendimiento: incluso si un branch casi siempre es **taken**, podemos predecir de forma incorrecta dos veces seguidas, en vez de una, cuando es **non-taken**, por ejemplo, en el caso de un *for* anidado:

```

1  for (i = 0; i < 256; i++)
2  {
3      for (j = 0; j < 256; j++)
4      {
5          ...
6      }
7  }

```

En este caso, se estaría fallando 512 veces. Para resolver este problema, se implementó un modelo de 2 bits con la propiedad de histéresis<sup>1</sup>, el cual cumple el siguiente diagrama de estados:



Figura 2.9: Branch Prediction Buffer - 2 bits

La idea es que no vamos a cambiar nuestra predicción al primer desacierto, sino que al segundo. Si tenemos 2 bits para predecir, podemos definir cuatro estados. El modelo predice taken para los estados 11 y 10, y predice non-taken para los estados 01 y 00. Hay dos niveles de predicción: el nivel fuerte y el nivel débil. El nivel fuerte ocurre cuando estamos prediciendo taken (o non-taken) todo el tiempo. Si predecimos **taken** y el resultado es non-taken, no cambiamos la predicción, sino que pasamos a un estado de predicción taken, pero débil. Si estamos en el estado débil y la siguiente predicción es taken, volvemos al estado taken con predicción fuerte, pero si por segunda vez **consecutiva** volvemos a obtener un resultado non-taken, pasamos a predecir non-taken con predicción fuerte.

Esto soluciona una cantidad muy grande de problemas en cuanto a la eficiencia de predicción de saltos. En la práctica, este tipo de Branch Predictor se implementa en la etapa de **Fetch** del pipeline

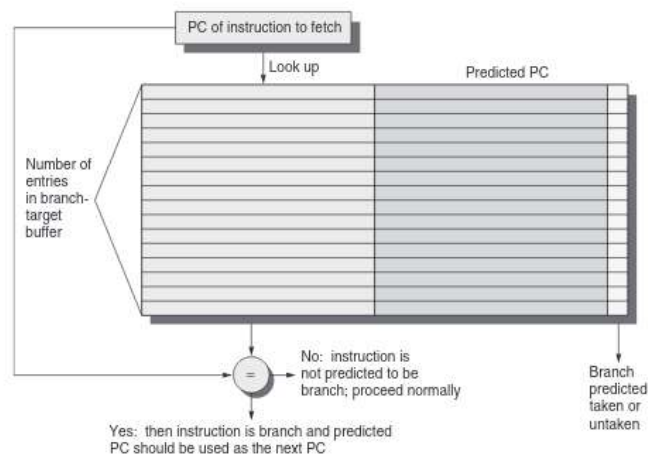
<sup>1</sup>tendencia de un material a conservar una de sus propiedades, en ausencia del estímulo que la ha generado

(para no perder tiempo), y se utiliza un pequeño cache de direcciones de salto accesible mediante las direcciones de las instrucciones (como vimos para el caso de 1 bit). También se puede implementar agregando un par de bits a cada bloque de líneas en el cache de instrucciones que se emplean únicamente si ese bloque tiene instrucciones de salto condicional.

### Branch Target Buffer

Hasta ahora, hemos asumido que el target de cada branch condicional era conocido, normalmente como una dirección explícita o como un desplazamiento relativo de la instrucción actual. A menudo, esta suposición es válida, pero algunas instrucciones de branch condicional calculan la dirección target haciendo aritmética en los registros. Incluso si se logra predecir con precisión cuándo saltar y cuándo no, tal predicción no sirve de nada si se desconoce la dirección de destino. Una forma de lidiar con esta situación es, en vez de solo almacenar bits que indican si fue taken o non-taken, almacenar la dirección a la cual se saltó la última vez.

Este diseño es conocido como **Branch Target Buffer**. Consiste en un cache de instrucciones de salto, donde se almacenan pares **dirección de la instrucción de salto : dirección target resuelta**. Es decir, no solo nos guardamos los 2 bits de estado<sup>2</sup>, sino que además nos guardamos directamente las direcciones target.



**Figure 3.21** A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

Figura 2.10: Branch Target Buffer

Este modelo trabaja como una memoria de acceso por contenido, y a diferencia del anterior, se accede mediante el valor completo del Program Counter (y no con los bits menos significativos).

Podemos definir el siguiente algoritmo para el manejo de predicciones cuando utilizamos un BTB:

- Si el valor no se encuentra, se asume **taken**.
  - Si el resultado es **non-taken**, se acepta el delay en el pipeline y no se almacenada nada en el BTB.
  - Si el resultado es **taken**, se ingresa el valor al BTB.
- Si el valor se encuentra en el BTB, se aplica la dirección target almacenada.
  - Si el resultado es **taken**, no hay penalidad y no se actualiza la BTB (porque ya tenemos el valor correcto).

<sup>2</sup>En el Pentium 4 estos bits de estado se guardan en una estructura conocida como **branch history table**.

- Si el resultado es **non-taken**, guarda el nuevo valor en el BTB (ya que el que está, no sirvió), luego de la penalidad correspondiente en el pipeline.

### 2.2.3. SPEC89

En la figura 2.11 podemos ver unos benchmarks de SPEC(89), el cual es uno de los consorcios más serios para lo que es benchmarking, al no responder a ninguna arquitectura en particular. SPEC es un consorcio en el que se juntan todos los actores de la industria: fabricantes de hardware, productores de compiladores, productores de software y las principales aplicaciones; y se testean no con programas hechos expresamente para hacer un benchmark, porque a lo mejor esos programas están hechos bajo un mecanismo mental de optimización para determinada arquitectura a la cual se está acostumbrado, volviéndose imparcial aún inconscientemente.

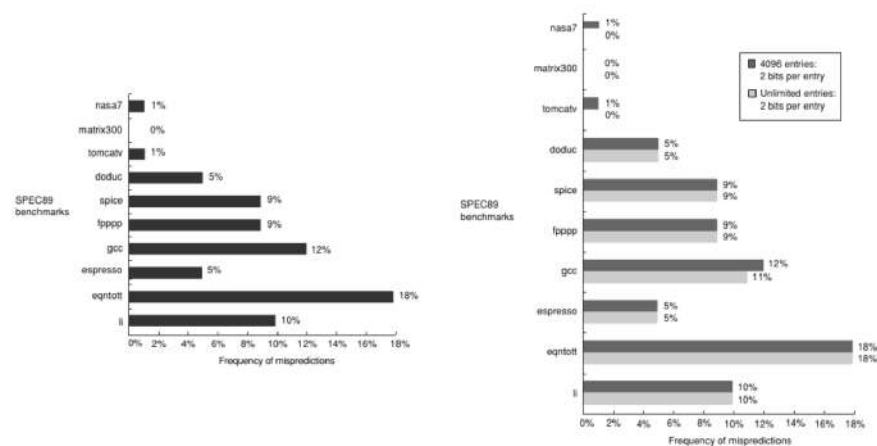


Figura 2.11: Benchmarks SPEC89

Lo que hacen acá es agarrar los fuentes de los programas donados por los miembros del consorcio, y meter *logs* de la información en determinados lugares. Miden hits en el cache, performance, cuántos ciclos de clock llevás hasta acá, cuántos saltos bien predichos tenés, cuántos saltos mal predichos tenés, etc. En particular, lo que vemos en la figura 2.11 es la frecuencia de saltos mal predichos para una predicción de salto de 2 bits, para el caso con una tabla de 4096 entradas, y para el caso de una tabla con muchas entradas (las suficientes para suponer ilimitadas).

A partir de esos resultados podemos concluir que

- El método tiene una eficiencia superior al 82 % para cualquier tipo de programa.
- La eficiencia es superior en programas de punto flotante ( $missprediction\ rate < 4\%$ ) frente a los programas de cálculo entero ( $4\% < missprediction\ rate < 18\%$ ).
- El tamaño del buffer de predicción no genera efecto en la eficiencia más allá de los 4KB.
- Tampoco se obtuvieron mejoras aumentando la cantidad de bits de predicción más allá de 2. En general, un branch predictor de  $n$  bits tomaría valores entre 0 y  $2^n - 1$ , tomando el salto para los valores contenidos en la mitad más alta del rango, y no lo tomaría para la mitad menos significativa. Pero, la complejidad en el diseño no se ve compensada por la mejora en la funcionalidad de predicción.

Entonces, utilizando esta máquina de estados la cosa funciona razonablemente bien, y ese efecto devastador del branch se da de una manera acotada y aceptable. Sin embargo, como el predictor de 2-bits solo utiliza el comportamiento reciente de un **único** branch para predecir el salto, es posible mejorar la precisión si tomamos en cuenta *otros* branches en vez de solo el que queremos predecir. El concepto clave es que muchas veces existe cierta correlación entre distintos branches, por lo que tomar en

cuenta los otros saltos nos aporta una mayor información que mirar cada branch de forma independiente. Consideremos el siguiente ejemplo:

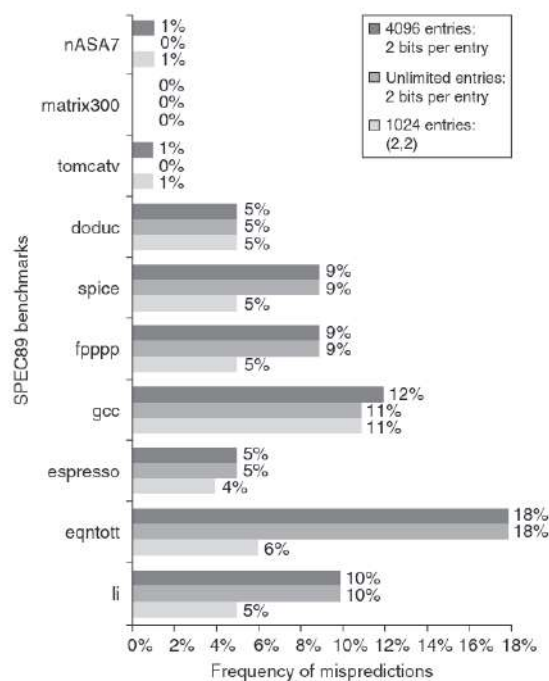
```

1  if (a == 2) a = 0;
2  if (b == 2) b = 0;
3  if (a != b) { ... }

```

Aquí podemos notar como el tercer branch está correlacionado con el comportamiento de los primeros branches. Si ambos branches resultan non-taken ( $a == 2$  y  $b == 2$ ), entonces sabemos que el tercer branch será taken ( $a == b$ ). Un predictor que solo haga uso de la información de cada branch por separado no podría capturar este comportamiento.

En la Fig. 2.2.3 podemos observar una comparación entre la performance entre tres predictores de saltos distintos: un predictor de 2-bits con 4096 entradas, un predictor de 2-bits con entradas ilimitadas y un predictor de correlación de 2-bits.



**Figure 3.3** Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

## 2.3. Arquitectura Superescalar

Hasta ahora veníamos trabajando con un único pipeline, pero ¿qué pasa si ponemos 2? Con 2 pipelines, en el extremo, vamos a ser capaces de obtener dos resultados por ciclo de clock.



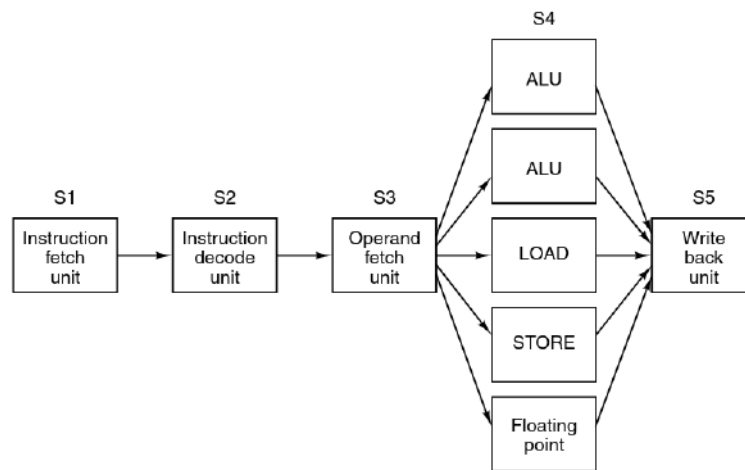
---

para instrucciones (code cache) y una cache para datos (data cache). Todos estos recursos se utilizaban intentando llegar a ese límite de dos instrucciones por ciclo de clock. Pero, ¿para qué pusieron un bus de datos de 64-bits?

La razón por la que el bus externo tenga 64-bits obedece a la necesidad de fetchear aún más rápido. Una forma de fetchear más rápido es ensanchar el bus de datos, de manera tal de poder fetchear varias instrucciones por cada acceso a memoria que hagamos, obteniendo un resultado equivalente a fetchear una instrucción por ciclo de clock.

Tenemos un bus interno de 256 bits entre la *code cache* y el buffer de pre-fetch, el cual busca reducir el cuello de botella generado por las instrucciones complejas de la ISA de Intel. Estas instrucciones pueden ser muy anchas o angostas, por lo que ponemos un bus de datos obscuramente grande, de manera tal que podemos mandar varias instrucciones a la vez.

Notemos que, si bien es posible seguir aumentando la cantidad de vías del pipeline hasta tener 2 o 4 vías, seguir aumentando la cantidad de pipelines duplicaría demasiado hardware de forma innecesaria. Luego, en la práctica, lo que se hace es tener uno o dos pipelines, pero tener múltiples unidades de ejecución, como se muestra en la figura 2.3. Esta idea se basa fuertemente en el hecho de que la tasa en la cual podemos preparar instrucciones para su ejecución es mucho más alta que la tasa de ejecución de las mismas.



**Figure 2-6.** A superscalar processor with five functional units.

Por último, al aumentar el paralelismo a nivel de instrucciones, aparecen más problemas, más obstáculos, más despelote. Los obstáculos estructurales quedan más expuestos que antes. Ahora, cada etapa, además de lidiar con las otras etapas de su propio pipeline, tiene que lidiar también con las mismas etapas del otro pipeline. Además, un miss en el *branch predictor* es letal, ya que se deben limpiar todos los pipelines. Por lo tanto, resolver todos estos problemas, de forma eficiente, se vuelve una tarea mortalmente importante.

## 2.4. Ejecución fuera de orden

La mayoría de las CPU modernas son superescalares y trabajan con pipelines. Típicamente estos procesadores cuentan con una cola FIFO de pre-fetching que nos permite buscar instrucciones antes de que se necesiten, para luego enviarlas a la unidad de decodificación. La unidad de decodificación envía las instrucciones decodificadas a las unidades funcionales adecuadas para su ejecución. En algunos casos, las instrucciones individuales se pueden dividir en micro-instrucciones antes de enviarlas a ejecutar.

Claramente, el diseño de una computadora sería más simple si todas las instrucciones se ejecutan en el orden en que se fetchean, es decir, respetando el orden especificado por el programa. Sin embargo, la ejecución en orden no siempre ofrece un rendimiento óptimo debido a las dependencias que pueden

---

existir entre las distintas instrucciones que un compilador no puede resolver. Habíamos visto que uno de los problemas que teníamos era que si, por ejemplo, una instrucción necesita un valor calculado por la instrucción anterior, la segunda no puede comenzar a ejecutarse hasta que la primera haya producido el valor necesario, y todas las demás instrucciones deben esperar (stall) a que se resuelva este obstáculo RAW.

En un intento por solucionar estos problemas y producir un mejor rendimiento, algunas CPU superescalares permiten reorganizar el orden de las instrucciones para acceder a instrucciones futuras que no son dependientes, produciendo los mismos resultados que si el programa se ejecutara en orden. Esta idea es conocida como **scheduling dinámico**.

### 2.4.1. Scheduling dinámico

El **scheduling dinámico** es la forma en la que el procesador organiza la ejecución de instrucciones, es decir, en qué orden las va mandando a las unidades de ejecución. Hasta este momento estábamos trabajando con pipelines con scheduling estático. En el modelo estático, se buscaba la instrucción, y luego se la enviaba a la unidad de ejecución (**dispatch**). En caso de tener dependencias con otra instrucción en el pipeline, el envío era demorado (**pipeline stall**).

En el caso de los superescalares, el estudio de dependencias se extiende a instrucciones en los diferentes pipelines. Es decir, no solo tenemos que mirar a las instrucciones dentro del pipeline, sino que además tenemos que mirar todas las instrucciones del resto de los pipelines. Por lo tanto, el método de **Forwarding** para reducir el efecto de los pipeline stalls se vuelve insuficiente, debemos utilizar soluciones más voladas.

Hasta ahora, si una instrucción  $j$  depende de una instrucción  $i$ , y la instrucción  $i$  requiere de varios ciclos de clock para completarse (debido a obstáculos), entonces la instrucción  $j$  y todas las instrucciones sucesoras no pueden ser ejecutadas y, por lo tanto, tenemos un **Pipeline stall**. Esta obstrucción del pipeline deja completamente inutilizado a todas las unidades funcionales que compongan a la Unidad de Ejecución de todos los pipelines.

Para ilustrarlo, consideremos el siguiente código de un procesador MIPS:

1	<code>div.d</code>	<code>F0, F2, F4</code>
2	<code>add.d</code>	<code>F10, F0, F8</code>
3	<code>sub.d</code>	<code>F12, F8, F14</code>
4	<code>mul.d</code>	<code>F6, F10, F8</code>

El problema acá es que la instrucción `div.d` es una instrucción iterativa, por lo que lleva varios ciclos de clock (8, 10 o 12 ciclos) para ejecutarse. Esto obstruye a la instrucción `add.d` que sí se puede resolver en un ciclo de clock, al no tener disponible el valor de `F0`.

Ahora bien, la instrucción `sub.d` no tiene ninguna dependencia con las previas, y sin embargo queda ahí esperando a que se complete la instrucción atascada. Estos escenarios son muy frecuentes, por lo que podemos preguntarnos ¿es necesario que el `sub.d` espere al `div.d`? ¿Qué pasaría si lo ejecutamos igual? Entramos, entonces, en lo que se conoce como **ejecución fuera de orden**

La idea es que podamos despachar las instrucciones a la unidad funcional correspondiente independientemente del orden en el que las instrucciones están especificadas en el programa, pero manteniendo un orden lógico determinado por las dependencias entre las distintas instrucciones, buscando obtener los mismos resultados, a pesar de alterar el ordenamiento de las instrucciones. La decisión de qué instrucción mandar a ejecutar se va a tomar en la etapa de decodificación, ya que ahí podemos saber si va a haber un obstáculo que impida el envío de las dependientes próximas. El hecho de que no respetemos el orden de las instrucciones implica que se nos van a presentar nuevas dependencias que antes no teníamos.

En general, tenemos tres tipos de órdenes importantes que ordenan el procesamiento de instrucciones:

- El orden en el que las instrucciones son fetcheadas.



- El orden en el que las instrucciones son ejecutadas.
- El orden en el que el resultado de las instrucciones es aplicado.

Cuanto más sofisticado el procesador, menores serán las relaciones estrictas entre estos órdenes. En todos los casos vamos a respetar el orden en que las instrucciones son fetcheadas, almacenándolas en una cola FIFO de pre-búsqueda. Sin embargo, el orden en que las instrucciones son ejecutadas y el orden en que sus resultados son aplicados puede ser alterado. En un pipeline con scheduling dinámico, vamos a ver que podemos ejecutar fuera de orden, aplicando los resultados en orden (reorder buffer) o fuera de orden (Scoreboarding/Tomasulo). Primero veamos qué ventajas traería la ejecución fuera de orden con aplicación de resultados en orden.

Supongamos que tenemos una máquina ideal, muy sencilla, que puede efectuar las etapas de **Fetch**, **Decode**, búsqueda de operandos, la fase de resultado (tomar el resultado de la ALU y guardarlo en algún lugar) y la fase de **Write** (aplicar el resultado donde corresponda) en 1 ciclo de clock. La etapa de ejecución va a depender de la instrucción utilizada: 5 ciclos de clock para la división, 1 para el resto.

¿Cuál es la necesidad de dividir la etapa de retiro en dos? Nuestro objetivo era avanzar en la ejecución de instrucciones, aplicando los resultados de las instrucciones en el mismo orden que está establecido en el programa.

Ahora, como las instrucciones están atomizadas en distintas etapas, lo que vamos a tratar de hacer es que parte de ese pipeline labure fuera de orden y el resto de las partes que laburen en orden. En este caso, lo único que vamos a manejar fuera de orden son las etapas de ejecución de las distintas instrucciones. En la figura 2.14 podemos ver la ejecución en orden y fuera de orden del código anterior.

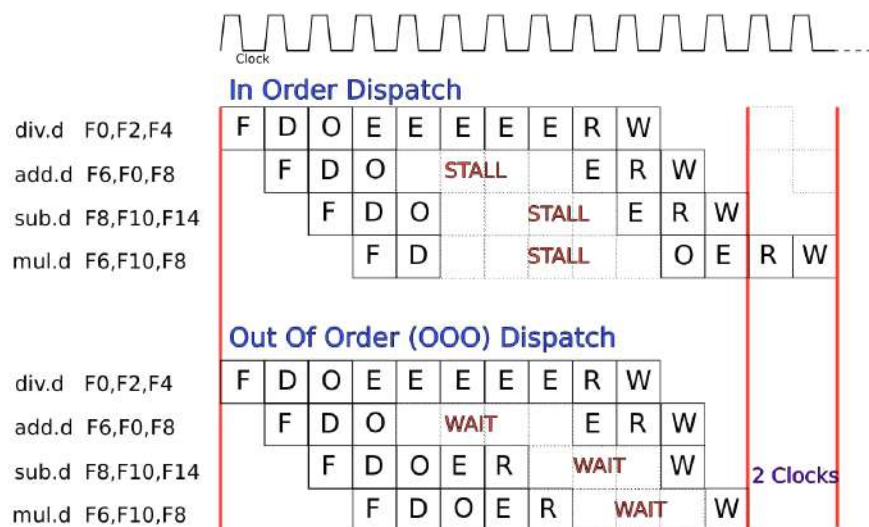


Figura 2.14: Fuera de Orden vs En Orden

En el caso de la ejecución en orden, el **add.d** tiene que esperar a que **div.d** termine de ejecutar, bloqueando al resto de las instrucciones, por lo que estamos en presencia de un **pipeline stall**.

Por otro lado, **sub.d** y **mul.d** pueden ejecutar sin ningún problema, pero deben esperar a que **add.d** termine de aplicar los resultados. Con esto, logramos reducir el tiempo que tardamos en obtener los resultados de 14 ciclos a 12 ciclos de clock, una mejora de más del 14 %.

Notemos que esta situación conlleva dos riesgos potenciales:

- La instrucción **sub.d** escribe su resultado en **F8**, el cual es un operando de **add.d**.
- La instrucción **mul.d** escribe su resultado en **F6**, el mismo operando destino que la instrucción **add.d**.



---

En general, el riesgo asociado a despachar la ejecución de la instrucción 3 se lo conoce como **WAR**. Si se la despacha, ésta *escribe* el registro F8, y *luego* la instrucción 2 *lee* F8, obteniendo un dato incorrecto.

Por otro lado, el riesgo asociado a lanzar la ejecución de la instrucción 4 se lo conoce como **WAW**. Si se despacha, ésta *escribirá* el registro F6, y *después* de ello la instrucción 2 lo *escribirá*, eliminando el resultado de la instrucción 4.

Además, tenemos al riesgo asociado a la dependencia de datos verdadera, se lo conoce como **RAW**, el cual aparece cuando la instrucción usa como operando un registro o dirección de memoria que es el resultado de la instrucción previa, generando un **pipeline stall**.

### 2.4.2. Manejando Excepciones

Ya vimos cómo podemos reducir el efecto de los branch penalties mediante la predicción de saltos y la ejecución fuera de orden. Sin embargo, cuando introducimos la posibilidad de ejecutar fuera de orden, aparecen nuevos problemas asociados a la dependencia de control entre las instrucciones.

Sabemos que alterar el orden en el que se ejecutan las instrucciones control dependientes podría romper con las dependencias de control. Para poder asegurar el correcto funcionamiento de los programas, debemos poder asegurar el cumplimiento de dos propiedades críticas: el comportamiento de excepción y el flujo de datos (que se tomen los branches correctos).

Preservar el comportamiento de excepción significa que cualquier cambio en el orden de ejecución de instrucciones no debe cambiar cómo las excepciones son generadas en un programa, es decir, no se deben generar nuevas excepciones por culpa del reordenamiento de las instrucciones. Consideremos el siguiente ejemplo:

```
1  add    R2, R3, R4
2  beqz   R2, go_to
3  load   R1, 0(R2)
4  go_to:
5  ...
```

En este caso, podemos ver que si no mantenemos la dependencia de datos que involucra R2, podríamos alterar el resultado del programa. Además, si ignoramos la dependencia de control y movemos la instrucción *load* antes del branch, podríamos acceder a una dirección inválida, generando una excepción. Notemos que ninguna dependencia de datos nos impide mover el *load* antes del *beqz*; solo nos lo impide la dependencia de control.

Ahora, recordemos cómo es que se atendían las excepciones, para luego ver qué problemas trae consigo la ejecución fuera de orden con aplicación de resultados fuera de orden. La implementación de las excepciones requiere que se llame a otro programa que se encargue de guardar el estado del programa en ejecución, corregir la causa de la excepción y luego restaurar el estado del programa antes de que se pueda volver a intentar ejecutar la instrucción que causó la excepción. Este proceso debe ser transparente para el programa en ejecución, por lo que es necesario que el pipeline proporcione al procesador herramientas para manejar la excepción, guardar el estado de la máquina y reiniciar sin afectar a la ejecución del programa.

Sin embargo, cuando trabajamos con ejecución fuera de orden, obtener el estado de la máquina deja de ser una tarea sencilla. Hay dos posibles motivos:

- El pipeline ha completado la ejecución de una o más instrucciones posteriores a la que produjo la excepción.
- El pipeline no ha completado aún alguna instrucción previa a la que generó la excepción.

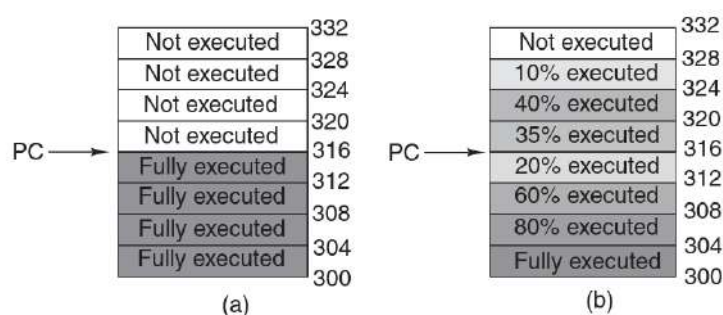
Esto quiere decir que si no tenemos cierto cuidado al momento de reordenar las distintas etapas en el procesamiento de instrucciones, no podríamos afirmar que todas las instrucciones hasta cierto punto fueron ejecutadas y que ninguna otra posterior fue ejecutada.

Típicamente, un procesador que quiera guardar el estado de la máquina debe asegurar que no se llame al handler de la excepción hasta haber completado de procesar todas las instrucciones anteriores a la responsable de la excepción. Es decir, tenemos que esperar a que la instrucción causante de la excepción le toque su turno en el programa original, como si se tratase de una máquina de ejecución secuencial. Cuando una excepción cumple con estas características, decimos que se trata de una máquina con excepciones **precisas**, característica que es deseable en una CPU (y obligatoria en algunas arquitecturas).

En resumen, podemos decir que una excepción es **precisa** si el estado guardado es consistente con la arquitectura secuencial del modelo. Es decir, se debe cumplir que

- Todas las instrucciones previas a la indicada por el Program Counter deben haber sido ejecutadas y deben haber modificado el estado del proceso correctamente.
- Ninguna de las instrucciones siguientes a la indicada por el Program Counter debe haber sido ejecutadas ni deben haber modificado el estado del proceso.
- El PC es guardado en un lugar conocido.
- El estado de ejecución de la instrucción apuntada por el PC es conocido.

La figura 2.4.2 ilustra el estado de la máquina en una excepción precisa (a) y en una excepción imprecisa (b).



**Figure 5-6.** (a) A precise interrupt. (b) An imprecise interrupt.

Una excepción que no cumple con estos requerimientos se dice **imprecisa**. Las máquinas con excepciones imprecisas suelen vomitar una gran cantidad de información sobre el estado interno al stack, para darle al handler la posibilidad de enterarse de lo que ocurrió. El código necesario para reiniciar la máquina suele ser demasiado complejo. Además, guardar grandes cantidades de información a la memoria cada vez que hay una excepción hace que estas sean lentas, y la recuperación incluso peor.

Las máquinas superescalares de la familia x86 tienen excepciones precisas, para lo cual utiliza una lógica de excepciones extremadamente compleja dentro de la CPU. Aquí el precio se paga no con tiempo, sino con área del chip que pudo ser utilizada para otra cosa. Algunas máquinas están diseñadas para tener algunas excepciones precisas y otras imprecisas. Por ejemplo, tener a interrupciones precisas de E/S, pero traps imprecisas, al no haber necesidad de reiniciar un proceso después de dividir por 0.

### 2.4.3. Algoritmo de Tomasulo

En 1967, Tomasulo presentó un trabajo sobre la implementación de scheduling dinámico en la FPU de la IBM 360/91. Su interés fue minimizar los riesgos **RAW** e implementar un método conocido como **Register Renaming**, utilizando buffers llamados **Reservation Stations**, con el objetivo de neutralizar los riesgos **WAR** y los **WAW**.

En la figura 2.15 podemos ver un esquema de la IBM/360 sin la mejora de Tomasulo. Tenía una unidad de instrucciones tipo pila (FLOS), buffers para almacenamiento (FLB), un sumador, un multiplicador, registros de punto flotante (FLR) y una unidad de almacenamiento de resultados (SDB).

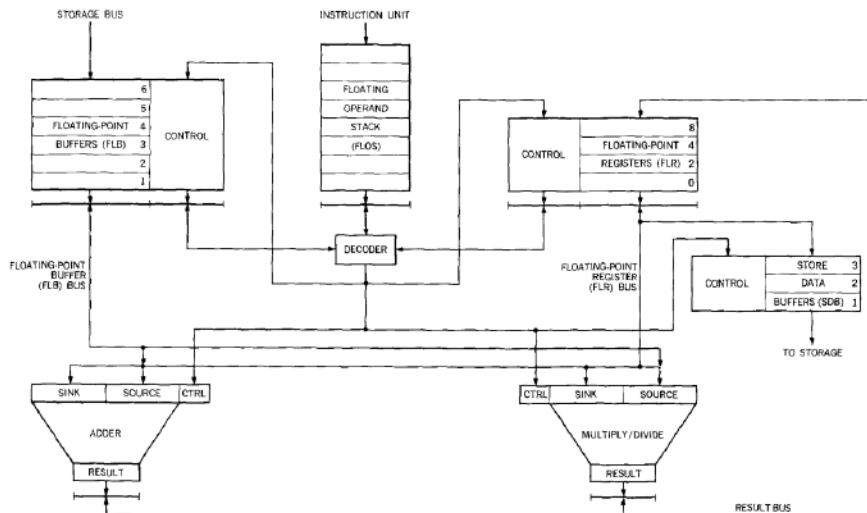


Figura 2.15: FPU IBM/360

Las instrucciones eran enviadas desde la unidad de instrucción hacia la cola de instrucciones, donde eran despachadas en orden FIFO. Los *load buffers* tienen dos funciones:

- Mantener los componentes de la dirección efectiva hasta que sea computada.
- Mantener los resultados de los loads completados que están esperando por el CDB.

Similarmente, los *store buffers* tienen dos funciones:

- Mantener los componentes de la dirección efectiva hasta que sea calculada.
- Mantener la dirección y valor a guardar hasta que la unidad de memoria esté disponible.

Lo primero que se preguntó Tomasulo es qué necesita un procesador para implementar la ejecución fuera de orden (**OOO**):

1. Lo primero que propuso fue mantener un enlace (link) entre el productor de un dato y con sus consumidores. El enlace entre el productor y los consumidores es crucial. La lógica del procesador tiene que tener claro para cada operando destino quién lo va a utilizar en las operaciones posteriores.
2. El segundo postulado nos dice que hay que mantener en espera a aquellas instrucciones que no estén listas para su ejecución. Esto se debe a que hasta que no estén todos los operandos disponibles, la instrucción no se puede enviar a ejecutar y, por lo tanto, debe haber algún lugar en donde podamos almacenar estas instrucciones.
3. Cada instrucción que está en espera debe saber cuándo están disponibles (*Ready*) sus operandos.
4. Finalmente, se despacha la instrucción a su Unidad Funcional cuando todos sus operandos estén *Ready*.

### Link productor-consumidor

¿Cómo producimos el enlace entre el productor del dato y sus consumidores? La solución de Tomasulo es el **Register Renaming**. La idea es asociarle a cada operando una etiqueta (*tag*) que permita identificar al registro de la RS que contiene a la instrucción que producirá su valor, el cual conseguimos de la RAT (Register Alias Table), que almacena el último tag de cada registro arquitectural. Consideremos el siguiente código (MIPS):

1	<code>div.d</code>	F0, F2, F4	
2	<code>add.d</code>	F6, F0, F8	# Riesgo WAR por F8 con sub.d
3	<code>store</code>	F6, 0(R1)	
4	<code>sub.d</code>	F8, F10, F14	

---

5	<code>mul.d</code>	F6, F10, F8	#	Riesgo WAW por F6 con <code>add.d</code> y Riesgo WAR con <code>s.d</code>
---	--------------------	-------------	---	--

Estas dependencias de tipo WAR y WAW pueden ser eliminadas utilizando *Register Renaming*. Consideremos dos registros no arquitecturales, que puedan ser utilizados como los registros de propósito general, a los que llamaremos *S* y *T*. Entonces, podemos utilizar estos registros auxiliares para reescribir el código anterior sin ninguna dependencia de tipo WAW ni WAR, de la siguiente manera:

1	<code>div.d</code>	F0, F2, F4
2	<code>add.d</code>	S, F0, F8
3	<code>store</code>	S, 0(R1)
4	<code>sub.d</code>	T, F10, F14
5	<code>mul.d</code>	F6, F10, T

Además, cualquier uso posterior de F8 debe ser reemplazado por T. El compilador, con algún soporte del hardware, debe "mirar más adelante" de cada instrucción en busca de riesgos. Esto demanda un poco más de sofisticación en la lógica y, consecuentemente, agrega complejidad tanto en el compilador como en el procesador (hoy está más volcado al lado del hardware).

En el esquema de Tomasulo, el *Register Renaming* fue el primer gran paso. La idea consiste en poder renombrar a los (pocos) registros arquitecturales asociándolos a (muchos) registros virtuales, en busca de minimizar los riesgos WAR y WAW. Este es provisto por las Reservation Stations (**RS**) y el Register Alias Table (**RAT**).

El **RAT** consistía en una tabla con una entrada por cada registro arquitectural, de manera tal que permita almacenar el último tag correspondiente a cada registro, para que la próxima instrucción a despachar sepa cómo referenciar a la instancia actual de cada uno de sus operandos. Las entradas de la RAT contenían tres campos: el campo *tag*, el campo *valor* y un bit de validez:

	Tag	Valor	Válido
R0			1
R1			1
R2			0
R3			1
R4			0
R5			1
R6			1
R7			0
R8			0
R9			0
R10			1
R11			0
R12			1
R13			1
R14			1
R15			0

Figura 2.16: RAT

Notemos que como el tag en el RAT es el último renombre del registro arquitectural, solo se aplicará el resultado asociado a ese tag, ya que todas las demás escrituras "ya pasaron", resolviendo los riesgos WAW.

Habíamos dicho que las instrucciones tenían que esperar en algún lugar a estar listas para ser despachadas. Ese lugar es un subsistema de hardware llamado **Reservation Station**, y es donde el procesador almacena todas aquellas instrucciones que no están *Ready*. En particular, vamos a tener el código de operación y para cada operando vamos a tener la misma estructura del *RAT*: un tag, un valor, y un bit de validez.

Hasta que el campo de validez no diga 1, ese operando no está *Ready*. Cuando todos los campos de validez de todos los operandos están en 1, la instrucción está *Ready* y, entonces, puede ser despachada. En definitiva, una **Reservation Station** termina siendo un gran buffer en el cual colocamos instrucciones. Cada operando, cuyo valor no esté disponible, posee un *tag* que se corresponde con el número del registro de la RS que se le asignó.

Cada vez que una Unidad de ejecución pone disponible un operando, es decir, cada vez que termina de ejecutar una instrucción, ese operando es un valor que va a ser consumido por otras instrucciones que se encuentran en la Reservation Station. O sea, ahora ese valor está asociado a un *tag*, porque como ese registro no tenía un valor válido, estaba renombrado con un *tag*. Entonces, ahora esa unidad de ejecución deberá informar a todo el sistema diciendo: "este *tag* ahora tiene este valor". Luego, todos los que tengan ese *tag* lo van a reemplazar con ese valor y van a marcar a ese operando como *Ready*. Es decir, pisan el valor, dejan el *tag* y ponen en 1 el bit de validez. Ahora, como el bit de validez está en 1, la lógica no va a mirar el *tag*, va a mirar el valor.

Cuando una instrucción tiene todos sus operandos disponible, la *Reservation Station* la despacha a la Unidad Funcional correspondiente (unidad de enteros, FPU, unidades de salto, etc). Si no hay ninguna unidad funcional disponible, la encola y espera a que se libere alguna. Si la *Reservation Station* tiene una cantidad muy superior de registros a la cantidad de registros de la arquitectura, potencialmente se pueden eliminar una gran parte de los riesgos estudiados (WAW, WAR).

En la figura 2.17 podemos ver el diseño de la IBM/360 con la mejora de Tomasulo. Podemos ver que se tiene una Reservation Station para la unidad sumadora de 3 registros, y otra para la unidad multiplicadora de otros 2 registros. Además, se agregó el Common Data Bus (CDB), el cual conecta a todo el sistema, permitiendo que cada vez que una de las unidades funcionales devuelva un resultado, esta pueda informarlo a todo el mundo vía este CDB.

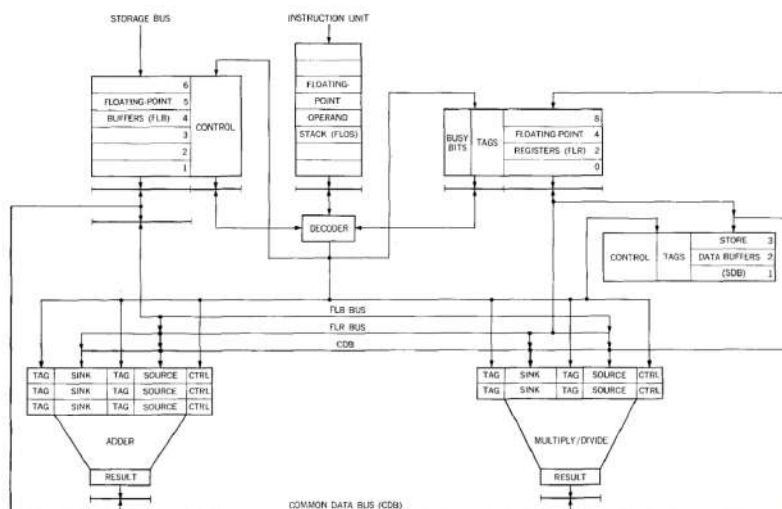


Figura 2.17: IBM/360 con la mejora de Tomasulo

El **CDB** resulta crucial para el *broadcast* de los resultados. En particular, cruza la salida de las unidades funcionales y atraviesa las *Reservation Station*, los *Floating Point Buffers*, los *Floating Point Registers* y el *Floating Point Operations Stack*.

Veamos más en detalle las distintas etapas por las cuales una instrucción debe pasar:

- **Fetch:** La instrucción es fetchada de la memoria de código, típicamente en bloques de dos o más instrucciones a la vez,
- **Issue:** El procesador determina a cuál RS debe despachar a la cabeza de la cola de instrucciones, basado en la unidad funcional que requiere la instrucción. Si hay una Reservation Station correspondiente a la instrucción que esté disponible, la instrucción es despachada a la misma, y

---

se actualiza el tag del registro destino en la RAT. Si no contamos con una, estamos frente a un obstáculo estructural y la instrucción se stallea hasta que se haya vaciado algún espacio en la RS.

Una instrucción puede ser despachada a pesar de que sus operandos no estén disponibles. Si algún operando no está disponible para leer desde el banco de registros (register file), esto seguramente se deba a que el valor asociado a ese registro todavía no fue calculado. En ese caso, el operando será actualizado con el resultado de una instrucción que ya había sido despachada (y asignada a una RS). Aquí entra en juego el tag que vincula al productor (la unidad funcional) con sus consumidores (las RS con instrucciones pendientes y el banco de registros). Retrasando la ejecución hasta que los operandos estén disponibles, los obstáculos de tipo RAW son evitados.

Mientras no tengamos un operando válido, vamos a estar tomando en cuenta el tag asociado al operando (obtenido de la RAT), y vamos a estar monitoreando por el CDB, esperando a que la unidad funcional que producirá el valor del operando envíe su tag y su valor (vía el CDB), una vez haya terminado de calcularlo. Notemos que esto vale para todas las Reservation Stations que esperaban ese operando.

- **Execute:** Cuando todos los operandos estén disponibles y la unidad funcional correspondiente está disponible para aceptar una nueva instrucción, la operación puede ser ejecutada. Sin embargo, como no tenemos ninguna estructura de HW que permita deshacer el efecto de ejecutar instrucciones, ninguna instrucción puede iniciar su ejecución hasta que todos los branches que la preceden sean completados. Esto reduce las ganancias de la predicción de saltos.
- **Write result:** Cuando el resultado esté disponible, enviamos el valor junto con el tag de la RS que produjo el resultado (a través del CDB), permitiendo entregar el resultado al registro destino (entrada de la RAT) y a las Reservation Stations que esperaban el resultado.

Es importante recordar que los tags en el esquema de Tomasulo hacen referencia a los registros de la RS que producirá el resultado, y el nombre original de los registros fuente son descartados cuando la instrucción se despacha a una Reservation Station.

El uso de Reservation Stations trae consigo dos propiedades importantes. En primer lugar, la detección de dependencias y el control de ejecución son distribuidos: la información almacenada en las Reservation Stations de cada unidad funcional determina cuándo una instrucción puede empezar la ejecución en esa unidad. En segundo lugar, todos los resultados son enviados a través del CDB, que se conecta directamente al banco de registros, a las Reservation Stations y a los store buffers, evitando tener que primero pasar por los registros, para luego avisar al resto de las unidades. De esta manera, en caso de que múltiples instrucciones estén esperando un solo resultado, y cada instrucción ya tenga su otro operando, entonces las instrucciones podrían ser enviadas a ejecutar de forma simultánea al momento de recibir el broadcast del resultado vía el CDB. Si se tuviese un banco de registros centralizado, las unidades necesitaría leer de los registros cuando los buses estén disponibles (que es lo que teníamos que hacer en el esquema de scoreboarding).

#### 2.4.4. Ejecución especulativa - Reorder Buffer

Volvamos al tema de predicción de saltos. A medida que aumentamos el paralelismo a nivel de instrucciones, el control de las dependencias del programa se complica proporcionalmente. La predicción de saltos nos permite reducir los stalls asociados a los branches, pero vimos que no podíamos ejecutar estas instrucciones hasta que sepamos la evaluación del branch, ya que no tenemos manera de deshacer la ejecución de los resultados. Esto resulta una pérdida considerable en el rendimiento para procesadores capaces de ejecutar múltiples instrucciones por clock. Luego, aumentar el paralelismo requiere de poder superar las limitaciones asociadas a las dependencias de control.

Es posible superar las dependencias de control al especular sobre la evaluación de los branches. Este mecanismo representa una extensión sutil, pero importante, de la predicción de saltos con scheduling dinámico. Cuando especulamos, fetchemos, despachamos y **ejecutamos** instrucciones, como si nuestras predicciones siempre fueran correctas. Lo más importante es tener la capacidad de deshacer la operación en caso de que la especulación no haya sido correcta.

La idea detrás de la implementación de especulación es permitir que las instrucciones sean ejecutadas

---

fuera de orden, pero forzarlas a que sean commiteadas en orden, evitando cualquier acción irreversible. De esta manera, se divide en dos la etapa de resultado: el broadcasteo de resultados entre las instrucciones ejecutadas de forma especulativa, y el comiteo de los resultados que se realiza una vez sepamos la evaluación del branch. Esto es esencial para el correcto funcionamiento del programa, ya que si no fuesen reversibles, podría pasar que estas instrucciones en realidad no debían haberse ejecutado, obteniendo resultados erróneos. Para implementar la ejecución especulativa, se introduce un módulo de hardware que resolvía este problema, conocido como **Reorder Buffer** (ROB).

El **ROB** suele ser un buffer circular con punteros de *head* y *tail*. Las entradas entre estos punteros son consideradas válidas<sup>3</sup>. Cuando una instrucción es despachada, la próxima entrada disponible en el ROB, apuntada por el *tail*, es asignada a la instrucción despachada. El número de la entrada asociado a esa instrucción termina siendo usado como *tag* para el renaming. Luego, el *tail pointer* es incrementado, módulo el tamaño del buffer. De esta manera, el ROB provee registros adicionales, donde se van almacenando los resultados de las instrucciones ejecutadas de forma especulativa, hasta que estos resultados sean comiteados. De esta manera, es posible ir aplicando los resultados en el orden original del programa, verificando la predicción de los branches. De esta manera, la ejecución se mantiene fuera de orden, pero la aplicación de resultados es en orden.

Por ejemplo, si la ventana tiene 8 instrucciones, y de esas 8 la primera no está lista, pero está lista la segunda y la tercera, no aplicamos nada, porque la primera no está lista y los resultados deben aplicarse en orden. Ahora, en cuanto la lógica nos marca como válida la primer instrucción, como la segunda y la tercera ya las tenemos listas, aplicamos las tres, y corremos la ventana tres lugares para abajo (actualizamos los punteros *head* y *tail*).

La diferencia con el algoritmo de Tomasulo es que este ponía el resultado en el registro directamente, y a partir de entonces el resto de las instrucciones lo tenía disponible. En cambio, cuando trabajamos con ejecución especulativa, el banco de registros no es actualizado hasta el commit de los resultados de las instrucciones, es decir, una vez tengamos la certeza de que esa instrucción es válida, y mientras tanto se trabaja de forma especulativa con los resultados que se fueron almacenando en el ROB. Es decir, el ROB se encarga de mantener la última versión de los registros, hasta que los resultados de las instrucciones anteriores sean commiteados al banco de registros.

Cada entrada del ROB contiene cuatro campos:

- El **tipo de instrucción**. Indica si la instrucción se trata de un branch (que no tiene operando destino), un store (que tiene como destino una dirección de memoria) o una operación en registros.
- El campo **destino**. Contiene el número del registro o de la dirección de memoria en donde se guardará el resultado.
- El campo **valor**. Es usado para mantener el resultado de la instrucción hasta el commit.
- El bit de **Ready**. Indica si la ejecución ya fue completada, indicando que el campo **valor** es válido.

A veces se cuenta con un **poison bit** que indica si la instrucción ejecutada de forma especulativa generó una excepción, e información asociada a esta. Esto se debe a que las instrucciones ejecutadas de forma especulativa podrían causar excepciones innecesarias. Luego de evaluar la condición del branch, podemos saber si esa excepción era o no necesaria, y recién ahí atenderla.

Además, es posible almacenar el Program Counter asociado a la instrucción, de manera tal que podamos volver hacia atrás en caso de tener una excepción o en caso de haber predicho de forma incorrecta un branch.

Ahora, veamos más en detalle las distintas etapas involucradas en la ejecución de una instrucción cuando tenemos un ROB:

- **Issue** (Envío): Se obtiene la primera instrucción de la cola de pre-búsqueda. La instrucción es despachada a una Reservation Station si tenemos una RS asociada a la unidad funcional corres-

---

<sup>3</sup>Si el *head* es mayor que el *tail*, las entradas válidas son desde el *head* hasta el final del buffer, y luego desde el inicio del buffer hasta el *tail*.

---

pendiente a la instrucción y si tenemos un espacio en el ROB. Si no hay RS disponible o no hay un slot libre en el ROB, hay un obstáculo estructural, con lo cual la instrucción queda temporalmente bloqueada (stall).

En el caso en sea despachada, hay que actualizar todas las estructuras internas con la información pendiente. Por ejemplo, si la escribe en una posición de la RS, tiene que ir a poner el tag en todos los lugares en los que se haga referencia al registro resultado de esa operación.

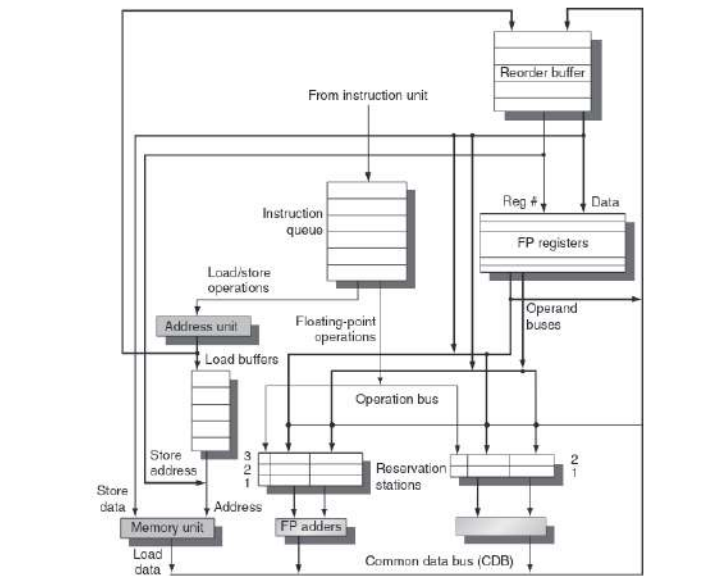
- **Ejecución:** Si uno o más operandos todavía no están disponibles, se monitorea por el CDB mientras que se espera a que estos sean broadcasteados. De esta manera se resuelven los riesgos RAW. Cuando ambos operandos están disponibles en la RS, se ejecuta la operación.
- **Write result:** Cuando el resultado está disponible, se escribe en el CDB junto con el tag (que se obtuvo del ROB), para broadcastear al resultado en el ROB y a las RS que esperaban el resultado, seteando el bit de validez para ese operando.
- **Commit:** Es la última fase, por lo que partir de aquí el resultado solo estará en el operando destino. Hay tres secuencias diferentes de eventos para esta etapa, dependiendo de si la instrucción es un branch con una predicción incorrecta, un store, o cualquier otra instrucción.
  - El caso de una instrucción normal ocurre cuando una llega a la cabeza del ROB y su resultado está presente. En este punto, el procesador actualiza el registro destino con el resultado y remueve la instrucción del ROB.
  - Si la operación es un memory store es similar, con la diferencia en que no se copia en el registro destino, sino que en la dirección de memoria (varios ciclos de clock después).
  - Cuando un branch con predicción incorrecta llega a la cabeza del ROB, esto indica que la especulación fue incorrecta. Por lo tanto, el ROB debe ser limpiado y la ejecución reiniciada en la correcta sucesora de la instrucción de branch.

Esto se debe a que si habíamos predicho que el salto iba a ser, por ejemplo, **non-taken**, el procesador siguió metiendo en los pipelines y las RS instrucciones que son las sucesoras secuenciales a la instrucción de salto. Sin embargo, al momento de evaluar la condición de salto nos dimos cuenta de que erramos la predicción. Por lo tanto, todo lo que tenemos en la RS y en el ROB que refieran a instrucciones posteriores a la instrucción de salto hay que tirarlo.

Si el ROB se llena, simplemente dejamos de despachar instrucciones hasta que una entrada esté disponible.

Por último, en la Fig. 2.18 podemos ver la estructura básica de una unidad de punto flotante utilizando el algoritmo de Tomasulo y una extensión para el manejo de ejecución especulativa.





**Figure 3.11** The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 3.6 on page 173, which implemented Tomasulo's algorithm, the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to multiple issue by making the CDB wider to allow for multiple completions per clock.

Figura 2.18: IBM/360 con la mejora de Tomasulo

## 2.5. Casos Prácticos

Ahora veamos algunos casos prácticos, así podemos relacionar todo lo visto con procesadores reales.

### 2.5.1. Three Cores Engine

En 1993 Intel presenta el procesador Pentium Pro. Este procesador representó una ruptura importante con el pasado. En lugar de tener dos o más pipelines, el Pentium Pro tenía una organización interna muy diferente y podía ejecutar hasta cinco instrucciones a la vez. Este modelo incluye una dos innovaciones importantes:

- Con este procesador se inaugura la Microarquitectura P6, en la cual se introducen los fundamentos de ejecución fuera de orden que se mantienen hasta el día de hoy.
- Otra innovación encontrada en el Pentium Pro fue una memoria cache de dos niveles. El chip del procesador en sí tenía 8 KB de memoria rápida para almacenar instrucciones y otros 8 KB de memoria rápida para almacenar datos. Además, se contaba con una segunda memoria cache de 256 KB (fuera del chip).

Más tarde fueron apareciendo nuevos modelos que trabajaban con la arquitectura P6, mejorando algunos aspectos:

- Pentium II: Aunque el Pentium Pro tenía un gran cache, carecía de las instrucciones MMX (porque Intel no podía fabricar un chip tan grande con rendimientos aceptables). Cuando la tecnología mejoró lo suficiente como para obtener tanto las instrucciones MMX como la cache en un solo chip, el producto combinado se lanzó como Pentium II.
- Pentium III: A continuación, se agregaron aún más instrucciones multimedia, llamadas SSE (Streaming SIMD Extensions), para mejores gráficos 3D. El nuevo chip se llamaba Pentium III, pero internamente era esencialmente un Pentium II.
- Celeron: la línea Celeron era básicamente una versión de bajo precio y bajo rendimiento del Pentium

---

2 destinada a las PC de gama baja.

- Xeon: la línea Xeon era básicamente una versión de gama alta del Pentium 2. Contaba con una cache más grande, un bus más rápido y un mayor soporte de multiprogramación.

Todos estos procesadores se basaron en una estructura interna que Intel denominó **Three Cores Engine**

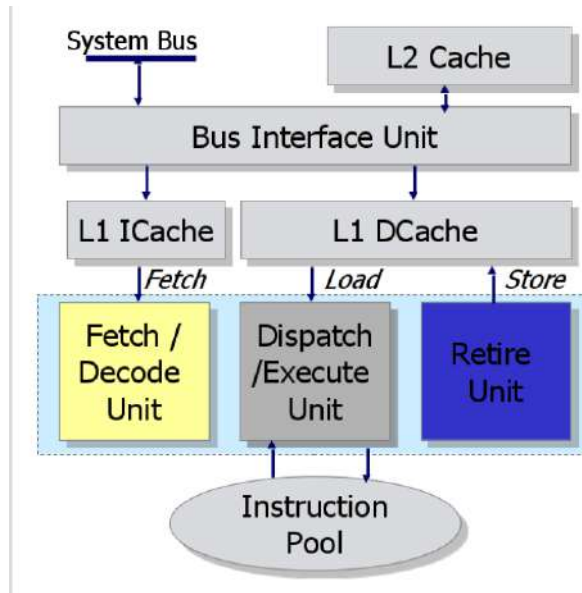


Figura 2.19: Intel - Three Cores Engine

donde los tres cores refiere a que tenemos una unidad de búsqueda de instrucciones y decodificación, una unidad de despacho y ejecución, y la unidad de retiro.

Notemos que trabajan con una arquitectura de tipo Harvard, ya que tenemos una L1 cache de instrucciones conectada al circuito de instrucciones y, por separado, una L1 cache de datos conectada al circuito de datos. Además, tenemos un bus para cargar y otro para escribir en la misma memoria, pudiendo leer y escribir datos en un mismo ciclo de clock, aumentando el paralelismo.

Este modelo emplea scheduling dinámico basado en una ventana de instrucciones, conocida como **Instruction Pool**. Este pool de instrucciones funcionaba como cinta transportadora de instrucciones, de manera tal que las tres unidades trabajen sobre la misma ventana de instrucciones.

Además instrucciones se traducen en micro-operaciones básicas ( $\mu$ OPs), y estas entran al Instruction Pool (vamos a ver que no es otra cosa que el Reorder Buffer), donde se mantienen para su ejecución especulativa. La idea de descomponer en micro-operaciones es que todas ellas se ejecuten en un tiempo igual y sean del mismo tamaño (esto facilita su almacenamiento en el ROB). Por último, la unidad de despacho y la unidad de ejecución mantienen un modelo superescalar y lo combinan con un súper pipeline de 20 etapas.

En la Fig. 2.20 podemos ver un esquema más detallado del Three Cores Engine.

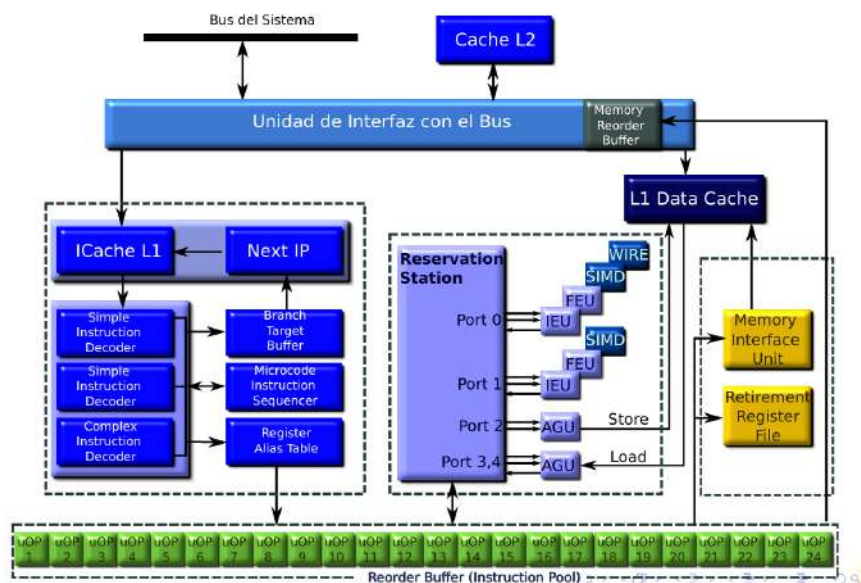


Figura 2.20: Three Cores Engine - en detalle

**Unidad de Interfaz con el Bus:** es un bloque que trabaja parcialmente en orden, y está encargado de conectar a los tres cores con el mundo exterior. Tiene un **Memory Reorder Buffer** que se encarga de ordenar lo que le llega desordenado del resto de los bloques. Se comunica en forma directa con el L2 cache (que está dentro del chip), buscando datos e instrucciones en ese cache. De alguna manera, esta interfaz divide la máquina en dos modelos: de la interfaz para afuera, la máquina es Von Neumann; mientras que de la interfaz para adentro, es Harvard. El bus que conecta la interfaz y el L2 cache es lo suficientemente ancho como para soportar hasta cuatro accesos concurrentes sobre el cache. El acceso al bus del sistema se controla mediante el protocolo MESI, permitiendo asegurar la coherencia de la memoria.

**Unidad de Fetch/Decode:** Tenemos un L1 cache de instrucciones (ICache), cuyo tamaño de línea es de 16 bytes (el mismo tamaño que el ancho del bus entre la interfaz y esta unidad). Se utilizaron 16 bytes porque las instrucciones de Intel podían llegar a ese ancho, no siempre, pero algunas son de ese ancho. Luego, esta cache le entrega 16 bytes alineados a la Unidad de Decodificación.

El **Next IP** apunta a la línea de la siguiente instrucción, ya sea la sucesora secuencial o a la target, según la predicción de salto generada en el **Branch Target Buffer**.

Hay tres decodificadores que toman la línea de 16 bytes entregada por el **ICache**. En particular, tenemos dos decodificadores de instrucciones simples y un decodificador de instrucciones complejas. Generalmente, los decodificadores simples traducen instrucciones a una única micro-operación, mientras que las instrucciones complejas se traducen en varias micro-operaciones. Cuando la instrucción es demasiado compleja, pasan por el **MicroCode Instruction Sequencer (MIS)**. El MIS termina funcionando como una ROM que toma una instrucción compleja y devuelve la secuencia de micro-operaciones decodificadas. De esta manera nos ahorramos tener una lógica iterativa ni secuencial.

Luego, las  $\mu$ ops se encolan y se envían al **Register Alias Table**, que se encarga de traducir los registros arquitecturales en registros virtuales, internos del procesador, implementando el **Register Renaming** de Tomasulo.

Por último, las  $\mu$ OPs ingresan al **Allocator**, donde se les agrega información de estado (bits de validez, etc) y se las agrega al pool de instrucciones.

**Unidad de Despacho/Ejecución:** se encarga de seleccionar las  $\mu$ OPs desde el pool de instrucciones, dependiendo de su **estado** y no de su orden dentro del pool de instrucciones. Entonces, la unidad de despacho es la única que actúa fuera de orden.

- Si el estado de la  $\mu$ OP indica que sus operandos están disponibles, la **Reservation Station** verifica que esté libre alguna unidad para su ejecución correspondiente, para enviarla a ejecutar.
- Una vez esté listo el resultado, este será escrito en el pool de  $\mu$ OPs por la Unidad de Ejecución correspondiente, y se setean los bits de validez correspondientes.
- Se dispone de 5 puertos de ejecución, de modo que se pueden programar a lo sumo 5  $\mu$ OPs por ciclo de clock (en la práctica se tiene un promedio de 3).
- Las  $\mu$ OPs de saltos se marcan como tales en el **ReOrder Buffer**, y se guarda junto a estas la dirección de salto predicha por el **BTB**. Si el salto falla, se limpian las entradas del **ROB** y en la **RS** correspondientes a todas las instrucciones posteriores a la del salto (incluso si están terminadas). Además, se modifica el **Next IP** de la unidad de decodificación.

**Unidad de Retiro:** se encarga de monitorear el estado de cada instrucción del **ROB**. No solo debe chequear su estado, sino que las debe reinsertar en el orden que tenían en el programa. Esto incluye el procesamiento de Interrupciones, excepciones, traps, breakpoints y predicciones fallidas.

Acá, Intel, hace algo que se llama **Retirement Register File (RRF)**, el cual se encarga de escribir los operandos resultantes en los registros de la arquitectura, mientras que la Unidad de Interfaz de Memoria se encarga de aplicar los resultados en el L1 cache de Datos (si el resultado corresponde a una dirección de memoria). Recordemos que el controlador cache (la Unidad de Interfaz con el Bus) se encarga de mantener la coherencia con el L2 Cache y la memoria principal.

### 2.5.2. Microarquitectura Netburst: Pentium 4

En la Fig. 2.21 podemos ver los bloques básicos de la microarquitectura Intel NetBurst del procesador Pentium 4. Podemos ver que contamos con cuatro secciones principales: el in-order front end, el out-of-order execution engine, las unidades de enteros y de punto flotante, y el subsistema de memoria.

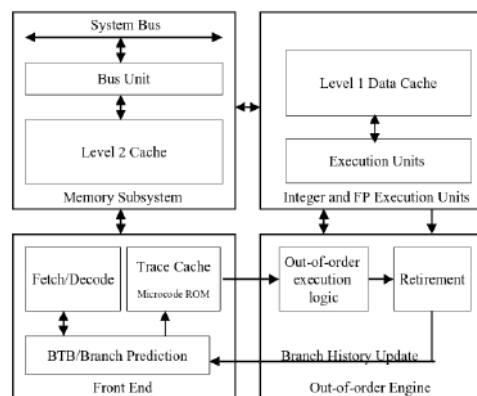


Figura 2.21: Intel Netburst

El front-end se encargaba de fetchear las instrucciones y prepararlas para su uso posterior en el pipeline. Su función es la de proveer instrucciones decodificadas al la unidad de ejecución. Se tiene una lógica de predicción de saltos que utiliza el historial de ejecución para especular sobre cuál será la próxima instrucción a ejecutar. En caso de que el branch no esté en el BTB, la predicción se realizará de forma estática basado en la dirección del desplazamiento del salto. Los saltos hacia atrás son asumidos **taken**, mientras que los saltos hacia adelante son asumidos **non-taken**.

Además, se cuenta con una Trace Cache en lugar de una L1 cache de código. La idea es que se guarden directamente las  $\mu$ ops, con el objetivo de reducir la cantidad de decodificaciones necesarias. De esta manera, las instrucciones eran decodificadas una única vez y colocadas en la Trace Cache, para luego ser utilizadas repetidas veces. Entonces, el decodificador solo debe ser utilizado cuando tenemos un miss en el Trace Cache y se requiere de ir al L2 cache para obtener la instrucción, y luego decodificarla. Esto

---

es particularmente útil considerando que las instrucciones de Intel son muy variadas, y por tanto difíciles de decodificar.

La L2 cache está conectada a la memoria principal, de manera tal que cuando tenemos un miss en la L2 cache o bien queremos acceder a los recursos de E/S, podamos acceder a la memoria principal a través del bus de sistema.

Una de las innovaciones en el Pentium 4 es que se podía duplicar la frecuencia del clock, con un PLL<sup>4</sup>, a ciertas partes del procesador. Por ejemplo, se tenían ALUs para instrucciones simples que trabajan con el doble de la frecuencia del clock, ya que estas se usan mucho más que las demás; y también se tenían ALUs para instrucciones más complejas que trabajan con una frecuencia menor. Por lo tanto, balancearon la velocidad en distintos valores, de manera tal que se mantuvo la potencia disipada en valores razonables.

El Out-of-Order execution engine se encarga de la reservación de registros, el register renaming y el scheduling de instrucciones. Esto permite la ejecución fuera de orden, a partir de renombrar a los registros arquitecturales, expandiendo de forma dinámica la cantidad de registros físicos del Pentium 4.

Se cuenta con una lógica de retiro que reordena las instrucciones, ejecutadas fuera de orden, en el orden original del programa. Además, asegura que las excepciones ocurran solo si la operación causante de la misma sea la instrucción más vieja, sin aplicar en la máquina (se preserva el comportamiento de excepción). Además, la lógica de ejecución se encarga de reportar las evaluaciones de los branches a los predictores de salto.

Además, para la versión de 3.06 GHz, se introdujo el uso de *hyperthreading*. Esta tecnología permitía que los programas dividan su trabajo en dos *threads* que el Pentium 4 podía procesar en paralelo. También se agregaron nuevas instrucciones SSE para mejorar el procesamiento de audio y video.

### 2.5.3. IA-64

Alrededor del año 2000, Intel estaba llegando al punto en que casi había exprimido hasta la última gota de jugo de la familia de procesadores IA-32. Los nuevos modelos seguían obteniendo beneficios de los avances en la tecnología de fabricación, lo que significa transistores más pequeños (por lo tanto, frecuencias de reloj más altas). Sin embargo, encontrar nuevos trucos para acelerar aún más la implementación se estaba volviendo cada vez más difícil a medida que las limitaciones impuestas por la arquitectura IA-32 eran cada vez mayores.

Antes de entrar en los detalles de la IA-64, es útil revisar qué está mal con la arquitectura IA-32 para ver qué problemas intentaba resolver Intel con la nueva arquitectura. La IA-32 es una ISA antigua con todas las propiedades incorrectas para la tecnología actual. Se trata de una arquitectura de tipo CISC con instrucciones de longitud variable y una gran cantidad de formatos diferentes que son difíciles de decodificar rápidamente sobre la marcha. La tecnología actual funciona mejor con arquitecturas de tipo RISC que tienen una longitud de instrucción y un código de operación de longitud fija que es fácil de decodificar. Las instrucciones IA-32 se pueden dividir en micro-operaciones similares a las RISC en tiempo de ejecución, pero hacerlo requiere hardware (área de chip), lleva tiempo y agrega complejidad al diseño.

Además, la mayoría de las instrucciones hacen accesos a memoria. La tecnología actual favorece las ISA de Load / Store que acceden a memoria solo para introducir los operandos en los registros, para luego realizar todos los cálculos utilizando instrucciones tres registros (como vimos en los procesadores MIPS). Considerando que la velocidad de la CPU aumenta mucho más rápido que la velocidad de la memoria, este el problema solo empeorará con el tiempo.

La IA-32 también tiene un conjunto de registros pequeño e irregular. La pequeña cantidad de registros de propósito general requiere que los resultados intermedios se almacenen en memoria todo el tiempo, provocando accesos innecesarios a memoria. Además, tener pocos registros aumenta la cantidad de dependencias, especialmente dependencias WAR. Además, en la arquitectura IA-32 se especifica el uso de interrupciones precisas. Todas estas cosas requieren una gran cantidad de hardware muy complejo.

---

<sup>4</sup>el PLL es un circuito que permite aumentar frecuencias

---

Hacer todo este trabajo rápidamente requiere un pipeline con muchas etapas. En consecuencia, una predicción de saltos muy precisa es esencial para asegurarse de que se ingresen las instrucciones correctas en el pipeline. Considerando que hay estudios que muestran que cada 20-30 instrucciones hay 5 saltos y que una predicción errónea requiere que el pipeline se vacíe, incluso una tasa de predicción errónea bastante baja puede causar una degradación sustancial del rendimiento. Para tratar de compensar las pérdidas en eficiencia asociadas a las predicciones incorrectas, el procesador tiene que hacer una ejecución especulativa, con todos los problemas que conlleva, especialmente cuando se acceden a branches erróneos que generan una excepción.

Una gran fracción de todos los transistores en los procesadores de la IA-32 están dedicados a descomponer las instrucciones CISC, descubrir qué se puede hacer en paralelo, resolver conflictos, hacer predicciones, reparar las consecuencias de predicciones incorrectas, dejando sorprendentemente pocos transistores para hacer el trabajo real.

Frente a este problema, se pensaba que la única solución real era abandonar el IA-32 como la línea principal de desarrollo e ir a una ISA completamente nueva. Esto es, de hecho, hacia lo que Intel comenzó a trabajar. En particular, se comenzó a trabajar en una nueva arquitectura, desarrollada conjuntamente por Intel y Hewlett-Packard, a la cual se la denominó **IA-64**.

Esta es una máquina completa de 64 bits de principio a fin con ejecución en paralelo, no una extensión de una máquina existente de 32 bits. Además, es una desviación radical de la arquitectura IA-32 en muchos sentidos. Sin embargo, no fue presentada adecuadamente al mercado tecnológico, por lo que fracasó. A pesar de esto, la arquitectura es tan radicalmente diferente de todo lo que hemos estudiado hasta ahora que vale la pena examinarlo solo por esa razón.

La idea clave detrás de la arquitectura IA-64 es mover el trabajo desde el tiempo de ejecución al tiempo de compilación. En el modelo IA-64, el compilador se encarga de reordenar instrucciones, renombrar registros, programar el uso de las unidades funcionales de antemano y producir un programa que se puede ejecutar como está, sin que el hardware tenga que hacer malabares con todo esto durante la ejecución. Es decir, que el compilador arme paquetes de instrucciones que se puedan ejecutar directamente en paralelo. El modelo de hacer visible el paralelismo subyacente en el hardware a los compiladores se conoce como **EPIC** (Explicitly Parallel Instruction Computing). Trabajando en esa línea, Intel, llegó a un modelo llamado **Itanium**, un procesador de 64-bits completamente diferente a lo que había.

Uno de los problemas que tenía el modelo IA-32 era el alto costo de acceder a memoria. Para evitar este problema, el modelo IA-64 tiene 128 registros de 64 bits de propósito general. Los primeros 32 de estos son estáticos, pero los 96 restantes se utilizan como una pila de registros. El número de registros visibles para el programa es **variable** y puede cambiar de un procedimiento a otro. Así, cada procedimiento tiene acceso a 32 registros estáticos y algún número (variable) de registros asignados dinámicamente.

Cuando se llama a un procedimiento, el puntero de la pila de registros avanza para que los parámetros de entrada sean visibles en los registros, pero no se asignan registros para las variables locales. El propio procedimiento decide cuántos registros necesita y avanza el puntero de la pila de registros para asignarlos. Estos registros no necesitan guardarse al entrar ni restaurarse al salir, aunque si el procedimiento necesita modificar un registro estático, debe tener cuidado de guardarlo explícitamente primero y restaurarlo más tarde. Al hacer que la cantidad de registros disponibles sea variable y se adapte a las necesidades de cada procedimiento, los escasos registros no se desperdician.

Además, también tiene 128 registros de punto flotante. Estos no funcionan como una pila de registros, pero al tener esta gran cantidad de registros, muchos cálculos de punto flotante pueden mantener todos sus resultados intermedios en registros y evitar tener que almacenar resultados temporales en la memoria.

También hay 64 registros de **predicado** de 1 bit, ocho branch registers y 128 registros de aplicaciones de propósito especial que se utilizan para diversos propósitos, como pasar parámetros entre programas de aplicación y el sistema operativo.

Otro de los principales problemas del modelo IA-32 es la dificultad de programar las diversas instrucciones en las distintas unidades funcionales evitando dependencias. El modelo IA-64 evita todos estos problemas haciendo que el compilador haga el trabajo. La idea clave es que un programa consta de una secuencia de **paquetes** de instrucciones. Dentro de ciertos límites, todas las instrucciones dentro de un

---

paquete no entran en conflicto entre sí, no usan más unidades y recursos funcionales que los que tiene la máquina, no contienen dependencias RAW y WAW, y solo tienen ciertas dependencias WAR restringidas. Como consecuencia de estas reglas, la CPU es libre de programar las instrucciones dentro de un grupo en el orden que elija, posiblemente en paralelo si puede, sin tener que preocuparse por los conflictos.

Otro de los problemas del modelo IA-32 es lidiar con los saltos condicionales. Si hubiera una forma de deshacerse de la mayoría de ellos, las CPU podrían volverse mucho más simples y rápidas. Al principio, podría parecer que deshacerse de los branches condicionales sería imposible porque los programas están llenos de *if*. Sin embargo, el modelo IA-64 usa una técnica llamada **predication** que puede reducir en gran medida su número.

En una arquitectura **predicada**, las instrucciones contienen condiciones (predicados) que indican cuándo deben ejecutarse y cuándo no. Este cambio de paradigma de instrucciones incondicionales a instrucciones predicadas es lo que nos permite deshacernos de (muchos) branches condicionales. En lugar de tener que elegir entre una secuencia de instrucciones incondicionales u otra secuencia de instrucciones incondicionales, todas las instrucciones se fusionan en una sola secuencia de instrucciones predicadas, utilizando diferentes predicados para diferentes instrucciones.

Las instrucciones predicadas se pueden introducir en el pipeline en secuencia, sin generar stalls ni otros problemas. Por eso son tan útiles. La forma en que realmente funciona la predicación en el IA-64 es que todas las instrucciones se ejecutan. Al final del pipeline, cuando llega el momento de retirar un resultado, se realiza una verificación para ver si el predicado era verdadero. Si es así, la instrucción se retira normalmente y sus resultados se vuelven a escribir en el registro de destino. Si el predicado es falso, no se realiza ninguna reescritura, por lo que la instrucción termina sin ningún efecto.

Además, las instrucciones predicadas pueden reordenarse, ya sea por el compilador o durante la ejecución (por ejemplo, para adelantar un LOAD). El único inconveniente es que la condición debe conocerse para cuando las instrucciones condicionales deban retirarse (cerca del final del pipeline).

El Itanium no calzó, entre otras cosas porque no era compatible con la arquitectura x86, y después el siguiente error fue que Intel vendía el compilador aparte, y como gcc no estaba pensado para arquitecturas EPIC, la arquitectura terminó siendo un fracaso. El Itanium fue diseñado para resolver las muchas deficiencias en la arquitectura IA-32. Dado que no fue ampliamente adoptado, ¿cómo abordó Intel las deficiencias del modelo IA-32? La clave no fue rearmar la ISA, sino adoptar la computación paralela a través de diseños de chips multiprocesador.

## 2.6. Procesadores Multithreads: Hyper-Threading

A principios de la década de 2000, procesadores como el Pentium 4 no ofrecían los aumentos de rendimiento que Intel necesitaba para mantener las ventas. Considerando que la propuesta de la arquitectura IA-64 fue un fracaso, se buscaron varias formas de aumentar el rendimiento sin cambiar la ISA. Las técnicas a nivel de microarquitecturas utilizadas para mejorar la eficiencia de un procesador incluyen el uso de un super-pipeline (un pipeline con muchas etapas atomizadas), predictores de salto, ejecución super-escalar, ejecución fuera de orden y el uso de varios niveles de caches. Estas técnicas hacen que los procesadores se volvieran cada vez más complejos, que tengan más transistores y que consuman más energía. Sin embargo, la velocidad en la que crece la cantidad de transistores y el consumo es mayor a la velocidad con la que mejora la eficiencia. Por lo tanto, se buscaba alguna manera adicional para mejorar aún más la performance, a un ritmo mayor al crecimiento de la cantidad de transistores y del consumo. La utilización de procesadores multi-threading es una solución.

Un vistazo al software de hoy en día revela que muchas aplicaciones consisten en múltiples threads o procesos que pueden ser ejecutados en paralelo. Agregando más procesadores, estas aplicaciones, potencialmente, podrían obtener mejoras sustanciales en su rendimiento a partir de la ejecución de múltiples threads en distintos procesadores al mismo tiempo. Estos threads pueden pertenecer a una misma aplicación, a otra aplicación corriendo en simultáneo, a un servicio del sistema operativo, o un thread del sistema operativo haciendo mantenimiento.

La idea básica del multithreading es permitir que dos threads (o posiblemente procesos, ya que la CPU no puede distinguirlos entre sí) se ejecuten a la vez en un mismo procesador en simultáneo,



haciendo un mejor uso de los recursos disponibles del procesador. Para el sistema operativo, un chip con multithreading parece un procesador dual en el que ambas CPU lógicas comparten una memoria cache y una memoria principal.

Esta idea estaba basada en el hecho de que no se estaba obteniendo mejoras en la performance a pesar de aumentar la cantidad de recursos disponibles. La explicación la tenemos en la figura 2.22. Supongamos una máquina con 5 unidades de ejecución, ahí vemos en cada ciclo de clock cuántas usa y cuántas no usa. Debido a instrucciones con dependencia de datos y a los saltos mal predichos, hay una buena parte de las unidades de ejecución que no se utilizan.

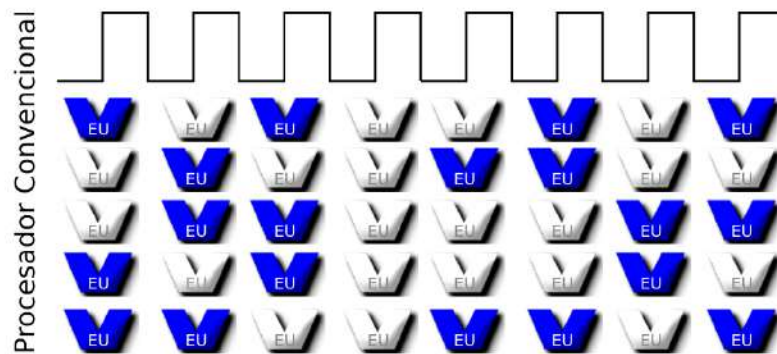


Figura 2.22: Ejecución en procesador convencional

La respuesta a este problema no era aumentar la cantidad de unidades de ejecución, sino ocuparlas. Entonces, la idea de multithreading es ocupar estas unidades de ejecución.



Figura 2.23: Ejecución en procesador multi-thread

Ahora, en el mismo tiempo se pueden procesar dos threads, donde el segundo thread se ejecuta en las unidades de ejecución libres.

La tecnología de Hyper-Threading implementa el multi-threading en los procesadores Intel. Para ello, se tiene un estado arquitectural para cada procesador lógico, y estos comparten un mismo conjunto de recursos físicos para la ejecución de instrucciones. Cada procesador lógico mantiene un estado arquitectural completo, el cual consiste en los registros de propósito general, los registros de control, los registros del APIC (Advanced Programmable Interrupt Controller), y otros registros asociados al estado de la máquina. Además, cada procesador tiene su propio APIC, lo cual permite que los threads puedan manejar las interrupciones de forma independiente. Notemos que la cantidad de transistores utilizados para almacenar el estado arquitectural es una pequeña fracción del total.



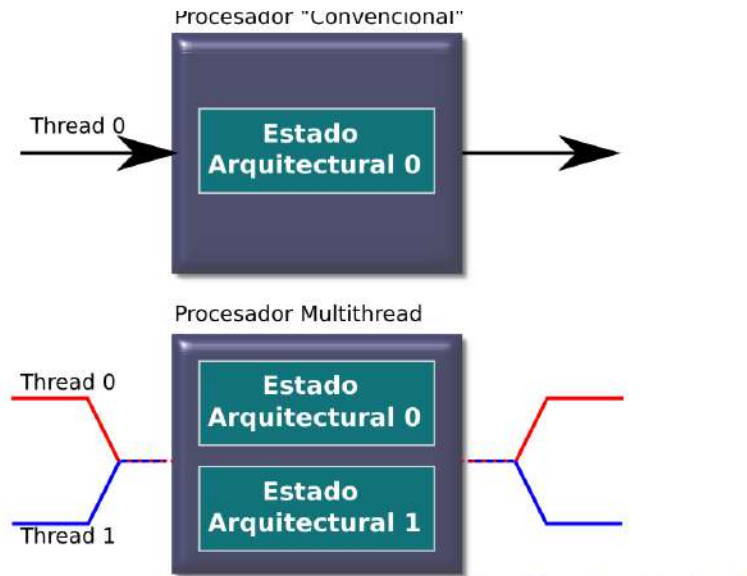


Figura 2.24: Procesador Multithread

Después de un poco de experimentación, quedó claro que un aumento del 5 % en el área del chip para el soporte de subprocesos múltiples dio una ganancia de rendimiento del 25 % en muchas aplicaciones, por lo que resulta en una buena alternativa. La primera CPU multithread de Intel fue la Xeon en 2002, pero posteriormente se agregó multithreading al Pentium 4 (hasta llegar a los Core i7 actuales). Intel llama a la implementación de multithreading utilizado en sus procesadores **hyperthreading**.

Sin embargo, utilizar multithreading también tiene sus desventajas. Si bien asignar recursos a cada thread es económico, compartir los recursos de forma dinámica requiere la contabilidad en tiempo de ejecución para monitorear el uso. Además, pueden surgir situaciones en las que los programas funcionen mucho peor con multithreading que sin él. Por ejemplo, si dos threads que necesitan 3/4 de la cache para funcionar bien, ejecutando por separado, cada uno funciona bien y se tiene un alto *hit-rate*. Si los ejecutamos juntos, cada uno tendría peor *hit-rate*, dando un resultado neto mucho peor que si no estuviéramos utilizando multithreading.

## 2.7. Procesadores Multicore

Si bien la estrategia de *multithreading* provee unas ganancias significantes en performance a un costo modesto, para algunas aplicaciones se necesitaban un rendimiento mucho mayor. Para esto, se desarrollan los chips **Multicore**.

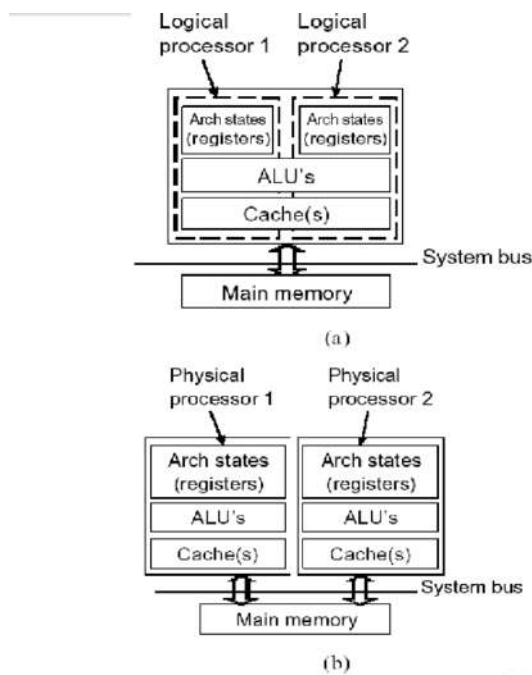


Figura 2.25: Hyperthreading vs Multicore

La idea era colocar múltiples procesadores (*cores*) en un mismo chip, que comparten una misma memoria principal. Como cada core puede leer o escribir cualquier parte de la memoria, tienen que coordinar, vía software, resolviendo conflictos sobre la memoria. Hay varios esquemas de implementación posibles, siendo el más sencillo conectar a las múltiples CPUs y a la memoria mediante un único bus (Fig 2.7(a)).

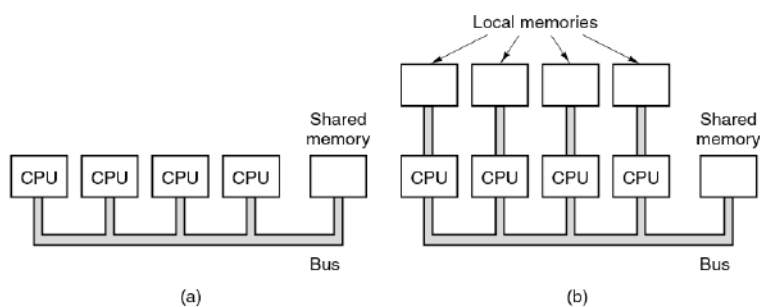


Figure 2-8. (a) A single-bus multiprocessor. (b) A multicomputer with local memories.

Para reducir la cantidad de conflictos sobre la memoria, es posible darle a cada procesador su propia memoria local privada, (Fig. 2.7(b)). Esta memoria, inaccesible para el resto de los procesadores, puede ser utilizada para código y datos que no sean compartidos. Acceder a la memoria privada evita tener que usar el bus principal, reduciendo notablemente el tráfico del mismo.

La pregunta que surgió entonces era ¿un *core* muy complejo o muchos *cores* sencillos? En el paper de 1995 "*Optimizing power using transformations*" se mostró que la potencia disipada de un procesador de dos *cores*, con la misma performance que un procesador de 1 core, era el 40 % de la potencia anterior.

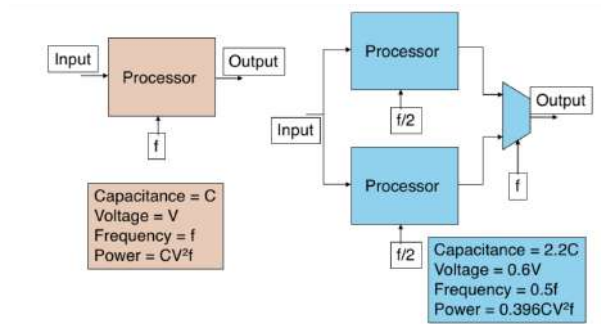


Figura 2.26: Varios cores: igual preformance, menor consumo

Por lo tanto, dos CPU en un chip consumen mucha menos energía que una CPU más compleja. Como consecuencia, la ganancia ofrecida por la ley de Moore puede explotarse cada vez más en el futuro para incluir más *cores* y caches en chip más grandes. Aprovechar estos multiprocesadores plantea grandes desafíos al desarrollo de software porque, a diferencia de las microarquitecturas de uniprosesor que podían extraer más rendimiento de los programas existentes, los multiprocesadores requieren que el software maneje explícitamente la ejecución en paralelo, usando *threads*, *semáforos*, memoria compartida, entre otras tecnologías.

## Capítulo 3

# Preguntas de Final

- **Coherencia de Cache:**

- Explicar cuándo empezamos a tener problemas con la coherencia y cuál es el problema con que la memoria esté incoherente respecto de las caches.
- Explicar las diferentes políticas de escritura, comparándolas según el uso de procesador y el uso del bus. ¿Cuál es más apta para un sistema Monoprocesador y cuál para un sistema SMP? Justificar.
- Explicar cómo se podría utilizar Copy Back en un sistema SMP.
- ¿Qué entiende por snooping y con qué elementos se implementa? ¿Cómo se complementa con el protocolo MESI? ¿Qué cosas se tienen que agregar a nivel de HW para implementar MESI (snoop bus, Shared, RFO)?
- En el protocolo MESI, ¿qué significa el estado "Modified"?
- MESI, tenés una línea en estado "Shared", ¿qué significa? ¿Qué pasa si la querés escribir? ¿Es impreciso?
- Si un procesador quiere leer una línea que él no tiene pero otro cache tiene en estado "Modified", ¿qué secuencia de cosas pasan?
- ¿Qué pasa si un procesador escribe en una línea con "Modified"? ¿Cómo afecta a la performance si se usa un protocolo con Write/copy back comparado con Write through?

- **Predicción de Saltos:**

- ¿Cómo funciona un predictor de saltos de 2 bits? Motivación y funcionamiento. Incluir diagrama y transiciones de estado.
- ¿En qué situaciones funciona bien un predictor de saltos de 2 bits y mal uno de 1 bit?
- ¿Por qué usar un predictor de 2 bits y no uno de 1 bit, 3 bits, spec89, etc?

- **Ejecución Fuera de Orden**

- Concepto y funcionamiento general. ¿Qué nuevas dependencias se introducen con la ejecución fuera de orden?
- Ventajas respecto de un esquema superescalar con ejecución en orden. Considerar que ambos modelos tienen la misma cantidad de vías de ejecución.

---

- **Algoritmo de Tomasulo**

- Explicar cuáles son los bloques de hardware que se agregan a un procesador superescalar, qué riesgos resuelve y cómo funciona cada uno.
- ¿Qué elementos tiene una Reservation Station?
- ¿Cómo se establece la relación consumidor/productor según Tomasulo? ¿Dónde está el tag o a qué hace referencia?
- ¿Cuándo se debe stallear una instrucción?
- Detallar secuencia de pasos para ejecutar una instrucción.

- **ReOrder Buffer**

- ¿Qué le faltó al algoritmo de Tomasulo para tener excepciones precisas?
- ¿Qué elementos tiene un reorder buffer?
- Explicar la implementación de Intel del Algoritmo de Tomasulo en el Three Cores Engine, detallando cada parte involucrada.

# Bibliografía

- [1] Herbert Bos Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson, 2014.
- [2] Todd Austin Andrew S. Tanenbaum. *Structured Computer Organization*. Pearson, 2013.
- [3] John L. Hennessy David A. Patterson. *Computer Organization and Design, Fifth Edition The Hardware Software Interface*. Morgan Kaufmann, 2013.
- [4] John Henessy & David Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufman, 2016.
- [5] William Stallings. *Computer Organization and Architecture: Designing for performance*. Pearson, 2016.